目錄

TensorFlow 教程

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

TensorFlow 教程 #01

简单线性模型

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube

中文翻译 thrillerist /Github

介绍

这份教程示范了在TensorFlow中使用一个简单线性模型的工作流程。在载入称为MNIST的手写数字图片数据集后，我们在TensorFlow中定义并优化了一个数学模型。（我们）会画出结果并展开讨论。

你应该熟悉基本的线性代数，Python和Jupyter Notebook编辑器。如果你对机器学习和分类有基本的理解也很有帮助。

导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.12.0-rc1'
```

载入数据

MNIST数据集大约有12MB，如果给定的地址里没有文件，它将自动下载。

```
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets("data/MNIST/", one_hot=True)
```

```
Extracting data/MNIST/train-images-idx3-ubyte.gz
Extracting data/MNIST/train-labels-idx1-ubyte.gz
Extracting data/MNIST/t10k-images-idx3-ubyte.gz
Extracting data/MNIST/t10k-labels-idx1-ubyte.gz
```

现在已经载入了MNIST数据集，它由70,000张图像和对应的标签（比如图像的类别）组成。数据集分成三份互相独立的子集。我们在教程中只用训练集和测试集。

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(data.train.labels)))
print("- Test-set:\t\t{}".format(len(data.test.labels)))
print("- Validation-set:\t{}".format(len(data.validation.labels)
))
```

```
Size of:
- Training-set:		55000
- Test-set:		10000
- Validation-set:	5000
```

One-Hot 编码

数据集以一种称为One-Hot编码的方式载入。这意味着标签从一个单独的数字转换成一个长度等于所有可能类别数量的向量。向量中除了第i个元素是1，其他元素都是0，这代表着它的类别是i'。比如，前面五张图像标签的One-Hot编码为：

```
data.test.labels[0:5, :]
```

```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.]])
```

在不同的比较和度量性能时，我们也需要用单独的数字表示类别，因此我们通过取最大元素的索引，将One-Hot编码的向量转换成一个单独的数字。需注意的是'class'在Python中是一个关键字，所以我们用'cls'代替它。

```
data.test.cls = np.array([label.argmax() for label in data.test.
labels])
```

现在我们可以看到测试集中前面五张图像的类别。将这些与上面的One-Hot编码的向量进行比较。例如，第一张图像的类别是7，对应的在One-Hot编码向量中，除了第7个元素其他都为零。

```
data.test.cls[0:5]
```

```
array([7, 2, 1, 0, 4])
```

数据维度

在下面的源码中，有很多地方用到了数据维度。在计算机编程中，通常来说最好使用变量和常量，而不是在每次使用数值时写硬代码。这意味着数字只需要在一个地方改动就行。这些最好能从读取的数据中获取，但这里我们直接写上数值。

```
# We know that MNIST images are 28 pixels in each dimension.
img_size = 28

# Images are stored in one-dimensional arrays of this length.
img_size_flat = img_size * img_size

# Tuple with height and width of images used to reshape arrays.
img_shape = (img_size, img_size)

# Number of classes, one class for each of 10 digits.
num_classes = 10
```

用来绘制图像的帮助函数

这个函数用来在3x3的栅格中画9张图像，然后在每张图像下面写出真实的和预测的类别。

```
def plot_images(images, cls_true, cls_pred=None):
    assert len(images) == len(cls_true) == 9

    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_shape), cmap='binary')

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i],
cls_pred[i])

        ax.set_xlabel(xlabel)

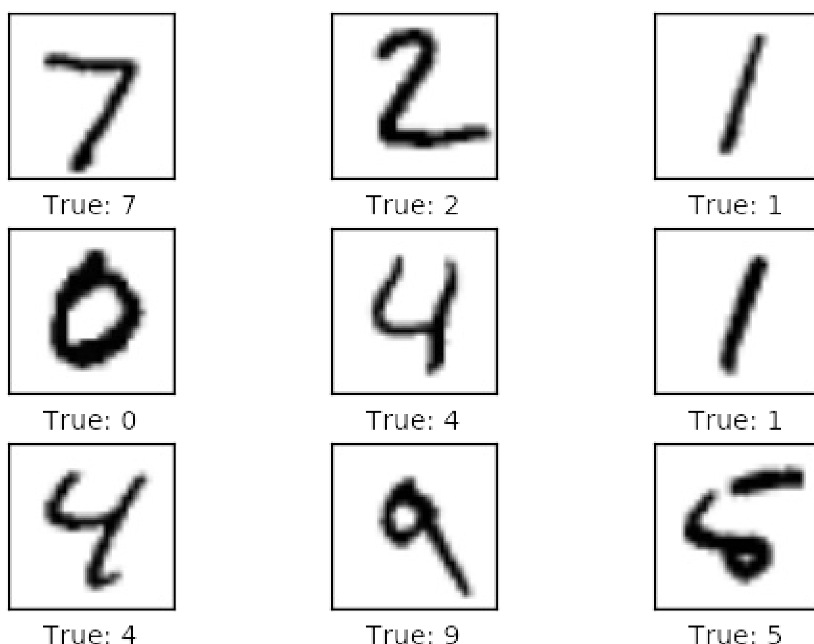
        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])
```

绘制几张图像来看看数据是否正确

```
# Get the first images from the test-set.
images = data.test.images[0:9]

# Get the true classes for those images.
cls_true = data.test.cls[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true)
```



TensorFlow图

TensorFlow的全部目的就是使用一个称之为计算图（computational graph）的东西，它会比直接在Python中进行相同计算量要高效得多。TensorFlow比Numpy更高效，因为TensorFlow了解整个需要运行的计算图，然而Numpy只知道某个时间点上唯一的数学运算。

TensorFlow也能够自动地计算需要优化的变量的梯度，使得模型有更好的表现。这是由于Graph是简单数学表达式的结合，因此整个图的梯度可以用链式法则推导出来。

TensorFlow还能利用多核CPU和GPU，Google也为TensorFlow制造了称为TPUs（Tensor Processing Units）的特殊芯片，它比GPU更快。

一个TensorFlow图由下面几个部分组成，后面会详细描述：

- 占位符变量（Placeholder）用来改变图的输入。
- 模型变量（Model）将会被优化，使得模型表现得更好。
- 模型本质上就是一些数学函数，它根据Placeholder和模型的输入变量来计算一些输出。
- 一个cost度量用来指导变量的优化。
- 一个优化策略会更新模型的变量。

另外，TensorFlow图也包含了一些调试状态，比如用TensorBoard打印log数据，本教程不涉及这些。

占位符（Placeholder）变量

Placeholder是作为图的输入，每次我们运行图的时候都可能会改变它们。将这个过程称为feeding placeholder变量，后面将会描述它。

首先我们为输入图像定义placeholder变量。这让我们可以改变输入到TensorFlow图中的图像。这也是一个张量（tensor），代表一个多维向量或矩阵。数据类型设置为 `float32` ，形状设为 `[None, img_size_flat]` ， `None` 代表tensor可能保存着任意数量的图像，每张图象是一个长度为 `img_size_flat` 的向量。

```
x = tf.placeholder(tf.float32, [None, img_size_flat])
```

接下来我们为输入变量 `x` 中的图像所对应的真实标签定义placeholder变量。变量的形状是 `[None, num_classes]` ，这代表着它保存了任意数量的标签，每个标签是长度为 `num_classes` 的向量，本例中长度为10。

```
y_true = tf.placeholder(tf.float32, [None, num_classes])
```

最后我们为变量 `x` 中图像的真实类别定义placeholder变量。它们是整形，并且这个变量的维度设为 `[None]` ，代表placeholder变量是任意长的一维向量。

```
y_true_cls = tf.placeholder(tf.int64, [None])
```

需要优化的变量

除了上面定义的那些给模型输入数据的变量之外，TensorFlow还需要改变一些模型变量，使得训练数据的表现更好。

第一个需要优化的变量称为权重 `weight` ，TensorFlow变量需要被初始化为零，它的形状是 `[img_size_flat, num_classes]` ，因此它是一个 `img_size_flat` 行、 `num_classes` 列的二维张量（或矩阵）。

```
weights = tf.Variable(tf.zeros([img_size_flat, num_classes]))
```

第二个需要优化的是偏差变量 `biases` ，它被定义成一个长度为 `num_classes` 的1维张量（或向量）。

```
biases = tf.Variable(tf.zeros([num_classes]))
```

模型

这个最基本的数学模型将placeholder变量 `x` 中的图像与权重 `weight` 相乘，然后加上偏差 `biases` 。

结果是大小为 `[num_images, num_classes]` 的一个矩阵，由于 `x` 的形状是 `[num_images, img_size_flat]` 并且 `weights` 的形状是 `[img_size_flat, num_classes]` ，因此两个矩阵乘积的形状是 `[num_images, num_classes]` ，然后将 `biases` 向量添加到矩阵每一行中。

```
logits = tf.matmul(x, weights) + biases
```

现在 `logits` 是一个 `num_images` 行 `num_classes` 列的矩阵，第i行第j列的那个元素代表着第i张输入图像有多大可能性是第j个类别。

然而，这是很粗略的估计并且很难解释，因为数值可能很小或很大，因此我们想要对它们做归一化，使得 `logits` 矩阵的每一行相加为1，每个元素限制在0到1之间。这是用一个称为softmax的函数来计算的，结果保存在 `y_pred` 中。

```
y_pred = tf.nn.softmax(logits)
```

可以从 `y_pred` 矩阵中取每行最大元素的索引值，来得到预测的类别。

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

优化损失函数

为了使模型更好地对输入图像进行分类，我们必须改变 `weights` 和 `biases` 变量。首先我们需要比较模型的预测输出 `y_pred` 和期望输出 `y_true` ，来了解目前模型的性能如何。

交叉熵（cross-entropy）是一个在分类中使用的性能度量。交叉熵是一个常为正值的连续函数，如果模型的预测值精准地符合期望的输出，它就等于零。因此，优化的目的就是最小化交叉熵，通过改变模型中 `weights` 和 `biases` 的值，使交叉熵越接近零越好。

TensorFlow有一个内置的计算交叉熵的函数。需要注意的是它使用 `logits` 的值，因为在它内部也计算了softmax。

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=l
ogits,
                                                         labels=y
_true)
```

现在，我们已经为每个图像分类计算了交叉熵，所以有一个当前模型在每张图上的性能度量。但是为了用交叉熵来指导模型变量的优化，我们需要一个额外的标量值，因此我们简单地利用所有图像分类交叉熵的均值。

```
cost = tf.reduce_mean(cross_entropy)
```

优化方法

现在，我们有一个需要被最小化的损失度量，接着我们可以创建优化器。在这种情况中，用的是梯度下降的基本形式，步长设为0.5。

优化过程并不是在这里执行。实际上，还没计算任何东西，我们只是往TensorFlow图中添加了优化器，以便之后的操作。

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.5)
.minimize(cost)
```

性能度量

我们需要另外一些性能度量，来向用户展示这个过程。

这是一个布尔值向量，代表预测类型是否等于每张图片的真实类型。

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

上面先将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确度。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
))
```

运行TensorFlow

创建TensorFlow会话（session）

一旦创建了TensorFlow图，我们需要创建一个TensorFlow session，用来运行图。

```
session = tf.Session()
```

初始化变量

我们需要在开始优化 `weights` 和 `biases` 变量之前对它们进行初始化。

```
session.run(tf.global_variables_initializer())
```

用来优化迭代的帮助函数

在训练集中有50,000张图。用这些图像计算模型的梯度会花很多时间。因此我们利用随机梯度下降的方法，它在优化器的每次迭代里只用到了一小部分的图像。

```
batch_size = 100
```

函数执行了多次的优化迭代来逐步地提升模型的 `weights` 和 `biases` 。在每次迭代中，从训练集中选择一批新的数据，然后TensorFlow用这些训练样本来执行优化器。

```
def optimize(num_iterations):
    for i in range(num_iterations):
        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = data.train.next_batch(batch_size
)

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        # Note that the placeholder for y_true_cls is not set
        # because it is not used during training.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimiz
er.
        session.run(optimizer, feed_dict=feed_dict_train)
```

展示性能的帮助函数

测试集数据字典被当做TensorFlow图的输入。注意，在TensorFlow图中，placeholder变量必须使用正确的名字。

```
feed_dict_test = {x: data.test.images,
                  y_true: data.test.labels,
                  y_true_cls: data.test.cls}
```

用来打印测试集分类准确度的函数。

```
def print_accuracy():
    # Use TensorFlow to compute the accuracy.
    acc = session.run(accuracy, feed_dict=feed_dict_test)

    # Print the accuracy.
    print("Accuracy on test-set: {0:.1%}".format(acc))
```

Function for printing and plotting the confusion matrix using scikit-learn.

用scikit-learn打印混淆矩阵。

```
def print_confusion_matrix():
    # Get the true classifications for the test-set.
    cls_true = data.test.cls

    # Get the predicted classifications for the test-set.
    cls_pred = session.run(y_pred_cls, feed_dict=feed_dict_test)

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_true,
                          y_pred=cls_pred)

    # Print the confusion matrix as text.
    print(cm)

    # Plot the confusion matrix as an image.
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)

    # Make various adjustments to the plot.
    plt.tight_layout()
    plt.colorbar()
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
    plt.yticks(tick_marks, range(num_classes))
    plt.xlabel('Predicted')
    plt.ylabel('True')
```

绘制测试集中误分类图像的函数。

```
def plot_example_errors():
    # Use TensorFlow to get a list of boolean values
    # whether each test-image has been correctly classified,
    # and a list for the predicted class of each image.
    correct, cls_pred = session.run([correct_prediction, y_pred_
cls],
                                    feed_dict=feed_dict_test)

    # Negate the boolean array.
    incorrect = (correct == False)

    # Get the images from the test-set that have been
    # incorrectly classified.
    images = data.test.images[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = data.test.cls[incorrect]

    # Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])
```

绘制模型权重的帮助函数

这个函数用来绘制模型的权重 `weights` 。画了10张图像，训练模型所识别出的每个数字对应着一张图。

```
def plot_weights():
    # Get the values for the weights from the TensorFlow variabl
e.
    w = session.run(weights)

    # Get the lowest and highest values for the weights.
    # This is used to correct the colour intensity across
    # the images so they can be compared with each other.
    w_min = np.min(w)
    w_max = np.max(w)

    # Create figure with 3x4 sub-plots,
    # where the last 2 sub-plots are unused.
    fig, axes = plt.subplots(3, 4)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Only use the weights for the first 10 sub-plots.
        if i<10:
            # Get the weights for the i'th digit and reshape it.
            # Note that w.shape == (img_size_flat, 10)
            image = w[:, i].reshape(img_shape)

            # Set the label for the sub-plot.
            ax.set_xlabel("Weights: {0}".format(i))

            # Plot the image.
            ax.imshow(image, vmin=w_min, vmax=w_max, cmap='seism
ic')

        # Remove ticks from each sub-plot.
        ax.set_xticks([])
        ax.set_yticks([])
```

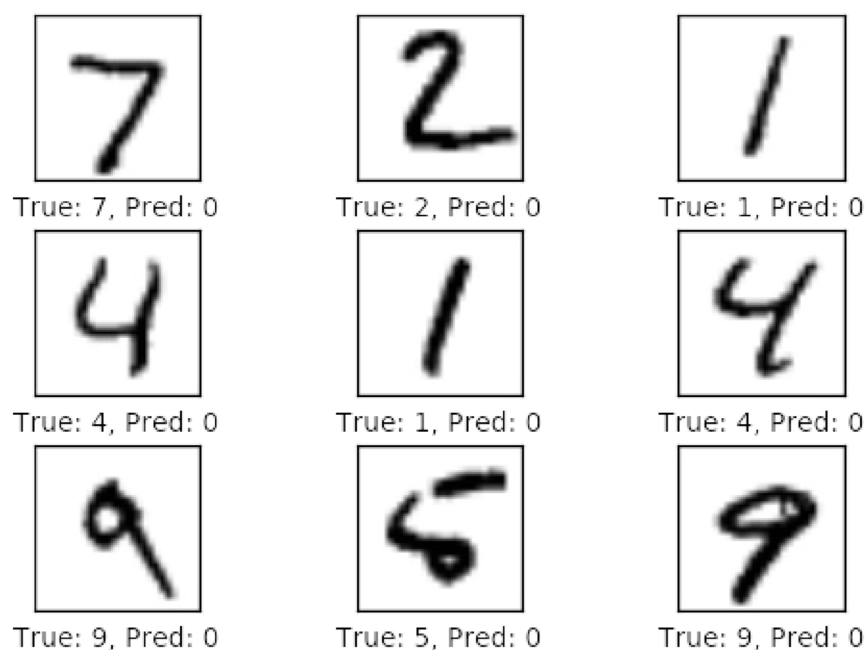
优化之前的性能

测试集上的准确度是9.8%。这是由于模型只做了初始化，并没做任何优化，所以它通常将图像预测成数字零，正如下面绘制的图像那样，刚好测试集中9.8%的图像是数字零。

```
print_accuracy()
```

```
Accuracy on test-set: 9.8%
```

```
plot_example_errors()
```



1次迭代优化后的性能

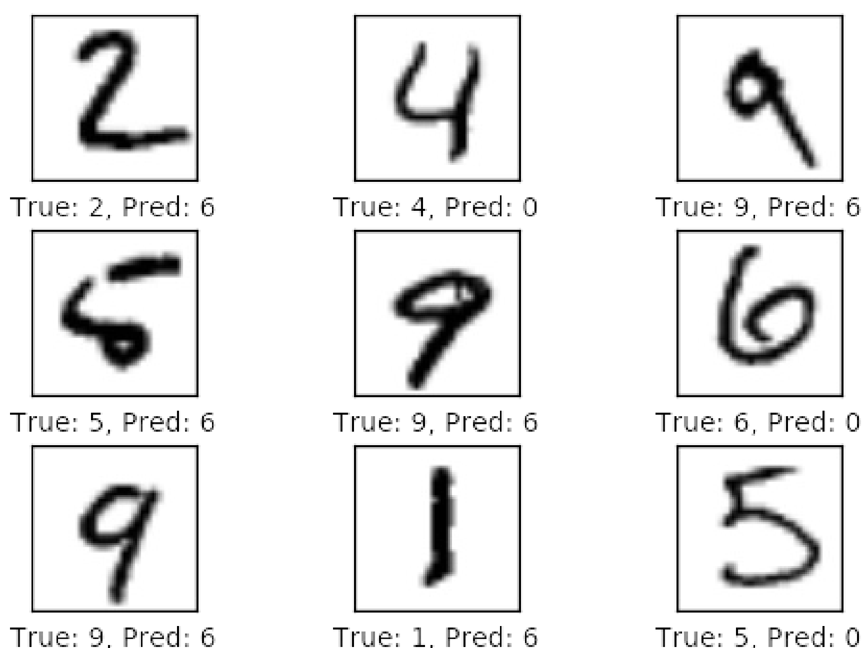
在完成一次迭代优化之后，模型在测试集上的准确率从9.8%提高到了40.7%。这意味着它大约10次里面会误分类6次，正如下面所显示的。

```
optimize(num_iterations=1)
```

```
print_accuracy()
```

```
Accuracy on test-set: 40.7%
```

```
plot_example_errors()
```



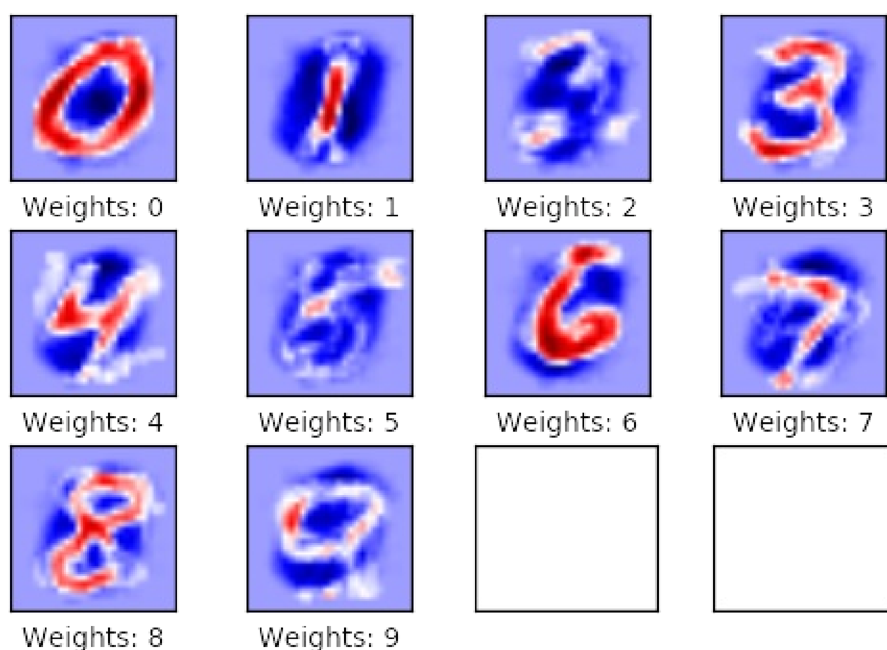
下面绘制的是权重。正值为红色，负值为蓝色。这些权重可以直观地理解为图像滤波器。

例如，权重用来确定一张数字零的图像对圆形图像有正反应（红色），对圆形图像的中间部分有负反应（蓝色）。

类似的，权重也用来确定一张数字一的图像对图像中心垂直线段有正反应（红色），对线段周围有负反应（蓝色）。

注意到权重大多看起来跟它要识别的数字很像。这是因为只做了一次迭代，即权重只在100张图像上训练。等经过上千张图像的训练之后，权重会变得更难分辨，因为它们需要识别出数字的许多种书写方法。

```
plot_weights()
```



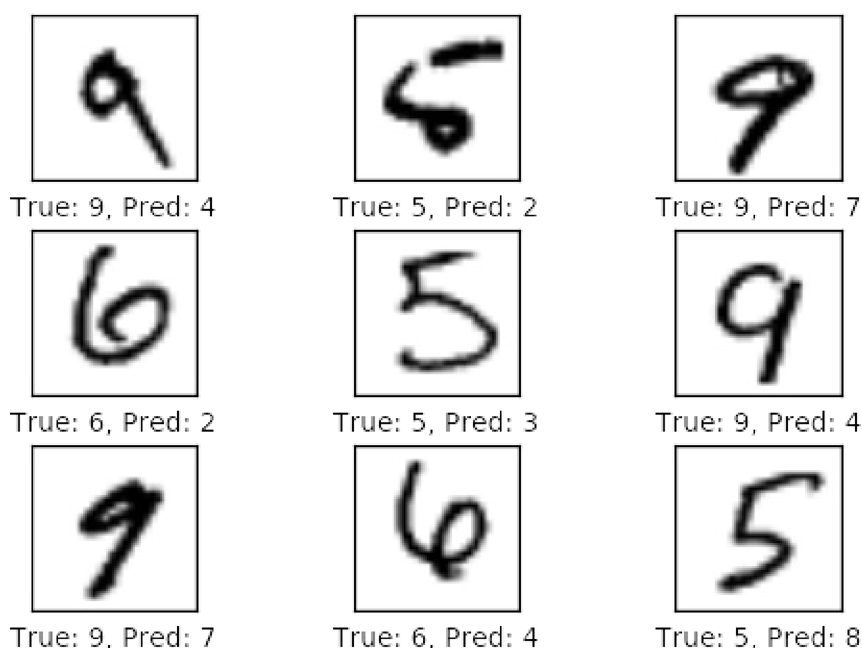
10次优化迭代后的性能

```
# We have already performed 1 iteration.
optimize(num_iterations=9)
```

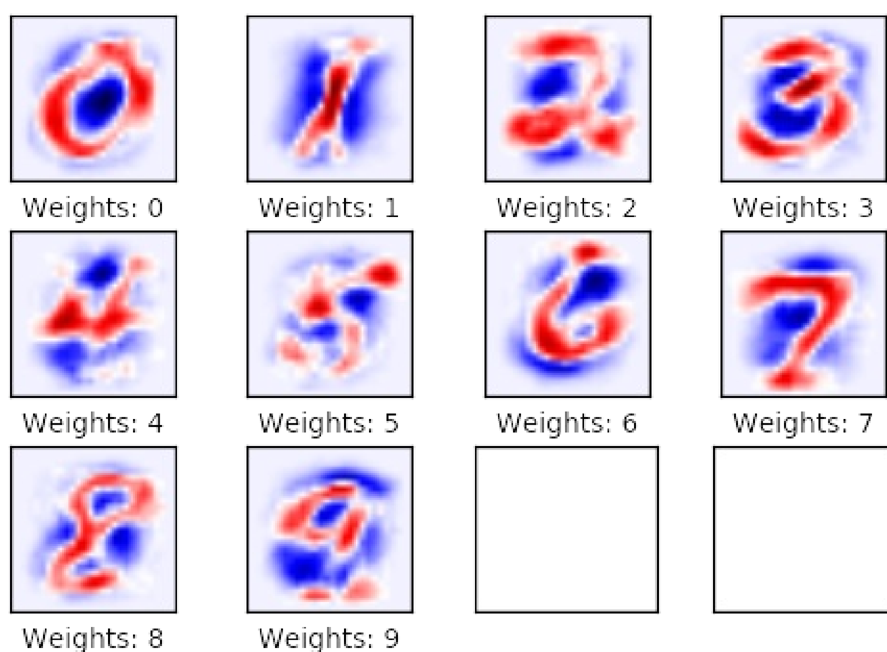
```
print_accuracy()
```

```
Accuracy on test-set: 78.2%
```

```
plot_example_errors()
```




```
plot_weights()
```



1000次迭代之后的性能

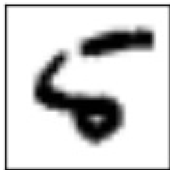
在迭代了1000次之后，模型在十次里面大约只误识别了一次。如下图所示，有些误识别情有可原，因为即使在人类眼里，也很难确定图像（的数字），然而有一些图像是很明显的，好的模型应该能分辨出来。但这个简单的模型无法达到更好的性能，因此需要更为复杂的模型。

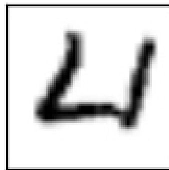
```
# We have already performed 10 iterations.
optimize(num_iterations=990)
```

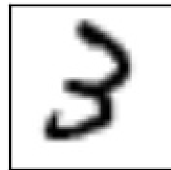
```
print_accuracy()
```

```
Accuracy on test-set: 91.7%
```

```
plot_example_errors()
```

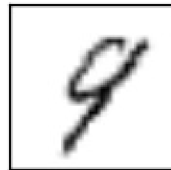

True: 5, Pred: 6

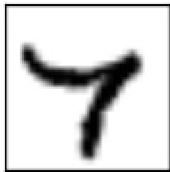

True: 4, Pred: 6

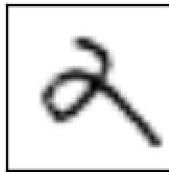

True: 3, Pred: 2

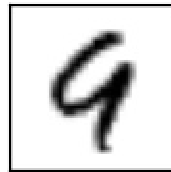

True: 9, Pred: 7


True: 2, Pred: 7


True: 9, Pred: 4

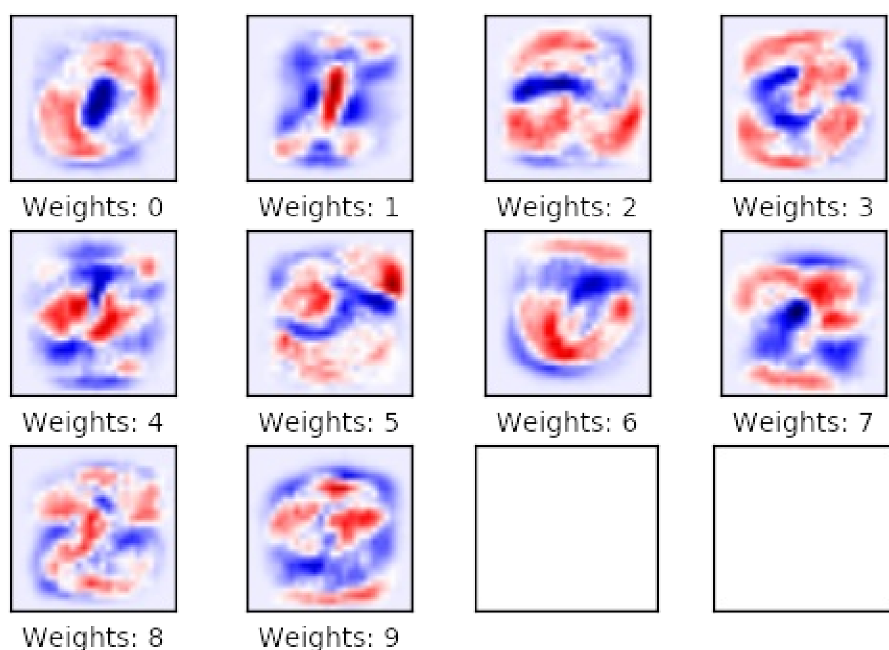

True: 7, Pred: 4


True: 2, Pred: 9


True: 9, Pred: 4

模型经过了1000次迭代训练，每次迭代用到训练集里面的100张图像。由于图像的多样化，现在权重变得很难辨认，我们可能会怀疑这些权重是否真的理解数字是怎么由线条组成的，或者模型只是记住了许多不同的像素。

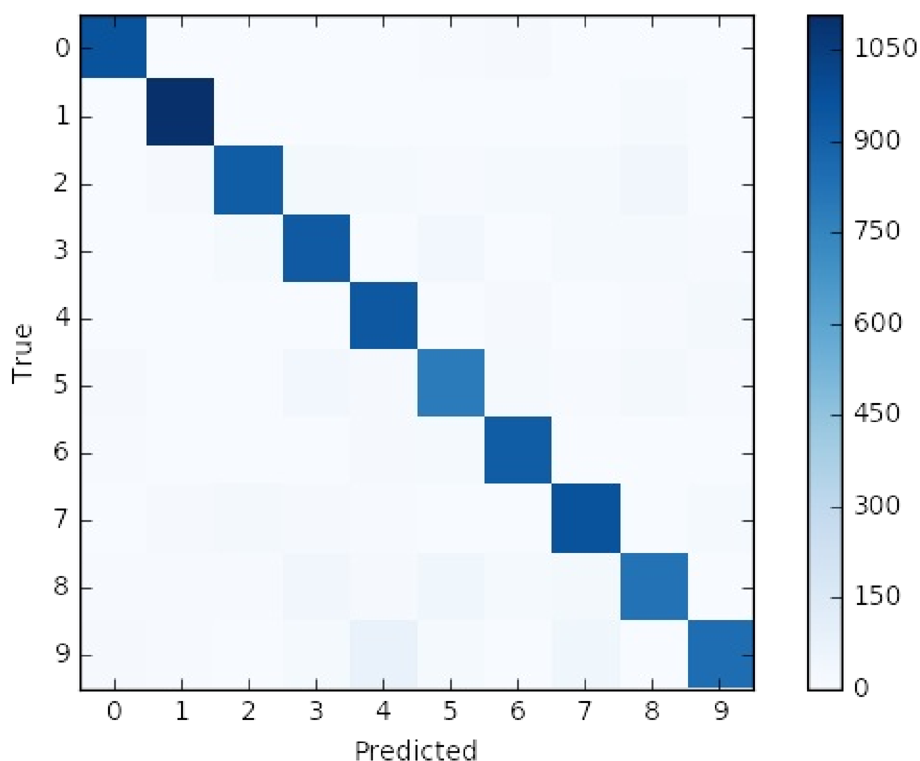
```
plot_weights()
```



我们也可以打印并绘制出混淆矩阵，它让我们看到误分类的更多细节。例如，它展示了描绘着数字5的图像有时会被误分类成其他可能的数字，但大多是3，6或8。

```
print_confusion_matrix()
```

```
[[ 957    0    3    2    0    5   11    1    1    0]
 [   0 1108    2    2    1    2    4    2   14    0]
 [   4    9  914   19   15    5   13   14   35    4]
 [   1    0   16  928    0   28    2   14   13    8]
 [   1    1    3    2  939    0   10    2    6   18]
 [  10    3    3   33   10  784   17    6   19    7]
 [   8    3    3    2   11   14  915    1    1    0]
 [   3    9   21    9    7    1    0  959    2   17]
 [   8    8    8   38   11   40   14   18  825    4]
 [  11    7    1   13   75   13    1   39    4  845]]
```



现在我们用TensorFlow完成了任务，关闭session，释放资源。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# session.close()
```

练习

These are a few suggestions for exercises that may help improve your skills with TensorFlow. It is important to get hands-on experience with TensorFlow in order to learn how to use it properly.

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。 在你对这个Notebook进行修改之前，可能需要先备份一下。

- 改变优化器的学习率。
- 改变优化器，比如用 `AdagradOptimizer` 或 `AdamOptimizer` 。
- 将batch-size改为1或1000。
- 这些改变如何影响性能？
- 你觉得这些改变对其他分类问题或数学模型有相同的影响吗?
- 如果你不改变任何参数，多次运行Notebook，会得到完成一样的结果吗？为什么？
- 改变 `plot_example_errors()` 函数，使它打印误分类的 `logits` 和 `y_pred` 值。

- 用 `sparse_softmax_cross_entropy_with_logits` 代替 `softmax_cross_entropy_with_logits` 。这可能需要改变代码的多个地方。探讨使用这两中方法的优缺点。
- 不看源码，自己重写程序。
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #02

卷积神经网络

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube

中文翻译 thrillerist/Github

介绍

先前的教程展示了一个简单的线性模型，对MNIST数据集中手写数字的识别率达到了91%。

在这个教程中，我们会在TensorFlow中实现一个简单的卷积神经网络，它能达到大约99%的分类准确率，如果你做了一些建议的练习，准确率还可能更高。

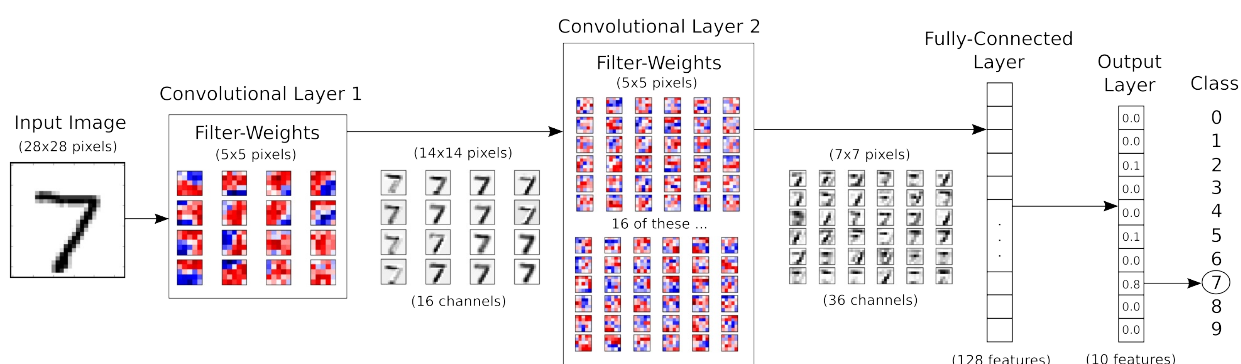
卷积神经网络在一张输入图片上移动一个小的滤波器。这意味着在遍历整张图像来识别模式时，要重复使用这些滤波器。这让卷积神经网络在拥有相同数量的变量时比全连接网络（Fully-Connected）更强大，也让卷积神经网络训练得更快。

你应该熟悉基本的线性代数、Python和Jupyter Notebook编辑器。如果你是TensorFlow新手，在本教程之前应该先学习第一篇教程。

流程图

下面的图表直接显示了之后实现的卷积神经网络中数据的传递。

```
from IPython.display import Image
Image('images/02_network_flowchart.png')
```



输入图像在第一层卷积层里使用权重过滤器处理。结果在16张新图里，每张代表了卷积层里一个过滤器（的处理结果）。图像经过降采样，分辨率从28x28减少到14x14。

16张小图在第二个卷积层中处理。这16个通道以及这层输出的每个通道都需要一个过滤权重。总共有36个输出，所以在第二个卷积层有16 x 36 = 576个滤波器。输出图再一次降采样到7x7个像素。

第二个卷积层的输出是36张7x7像素的图像。它们被转换到一个长为7 x 7 x 36 = 1764的向量中去，它作为一个有128个神经元（或元素）的全连接网络的输入。这些又输入到另一个有10个神经元的全连接层中，每个神经元代表一个类别，用来确定图像的类别，即图像上的数字。

卷积滤波一开始是随机挑选的，因此分类也是随机完成的。根据交叉熵（cross-entropy）来测量输入图预测值和真实类别间的错误。然后优化器用链式法则自动地将这个误差在卷积网络中传递，更新滤波权重来提升分类质量。这个过程迭代了几千次，直到分类误差足够低。

这些特定的滤波权重和中间图像是一个优化结果，和你执行代码所看到的可能会有所不同。

注意，这些在TensorFlow上的计算是在一部分图像上执行，而非单独的一张图，这使得计算更有效。也意味着在TensorFlow上实现时，这个流程图实际上会有更多的数据维度。

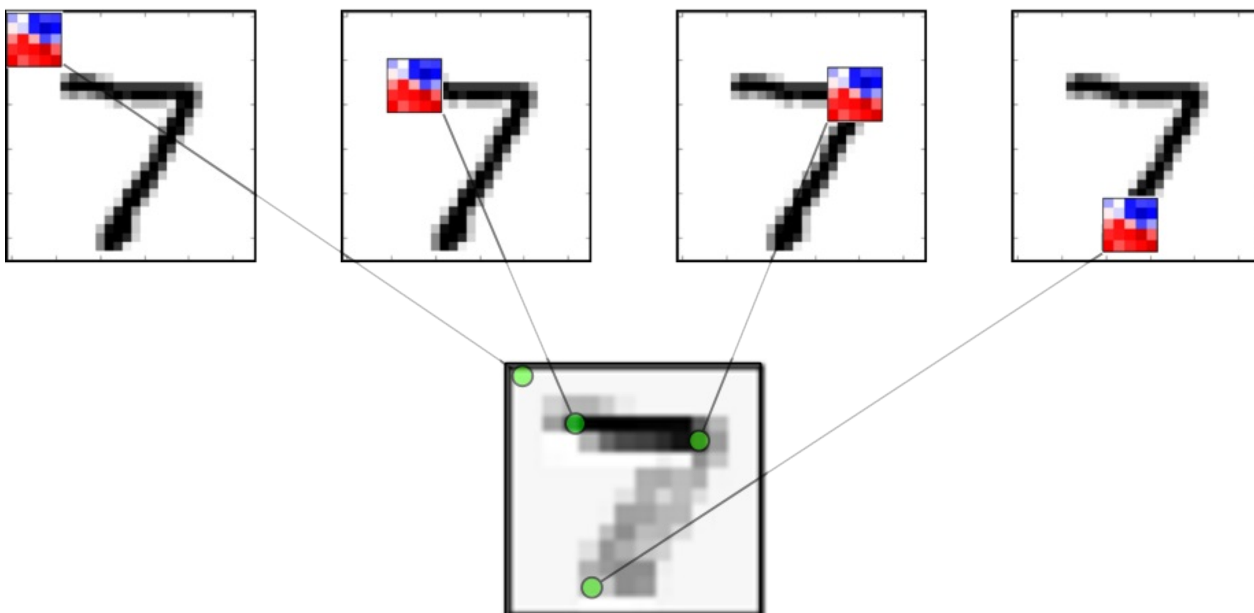
卷积层

下面的图片展示了在第一个卷积层中处理图像的基本思想。输入图片描绘了数字7，这里显示了它的四张拷贝，我们可以很清晰的看到滤波器是如何在图像的不同位置移动。在滤波器的每个位置上，计算滤波器以及滤波器下方图像像素的点乘，得到输出图像的一个像素。因此，在整张输入图像上移动时，会有一张新的图像生成。

红色的滤波权重表示滤波器对输入图的黑色像素有正响应，蓝色的代表有负响应。

在这个例子中，很明显这个滤波器识别数字7的水平线段，在输出图中可以看到它对线段的强烈响应。

```
Image('images/02_convolution.png')
```



滤波器遍历输入图的移动步长称为stride。在水平和竖直方向各有一个stride。

在下面的源码中，两个方向的stride都设为1，这说明滤波器从输入图像的左上角开始，下一步移动到右边1个像素去。当滤波器到达图像的右边时，它会返回最左边，然后向下移动1个像素。持续这个过程，直到滤波器到达输入图像的右下角，同时，也生成了整张输出图片。

当滤波器到达输入图的右端或底部时，它会用零（白色像素）来填充。因为输出图要和输入图一样大。

此外，卷积层的输出可能会传递给修正线性单元（ReLU），它用来保证输出是正值，将负值置为零。输出还会用最大池化(max-pooling)进行降采样，它使用了2x2的小窗口，只保留像素中的最大值。这让输入图分辨率减小一半，比如从28x28到14x14。

第二个卷积层更加复杂，因为它有16个输入通道。我们想给每个通道一个单独的滤波，因此需要16个。另外，我们想从第二个卷积层得到36个输出，因此总共需要16 x 36 = 576个滤波器。要理解这些如何工作可能有些困难。

导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
import time
from datetime import timedelta
import math
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.12.0-rc0'
```

神经网络的配置

方便起见，在这里定义神经网络的配置，你可以很容易找到或改变这些数值，然后重新运行Notebook。

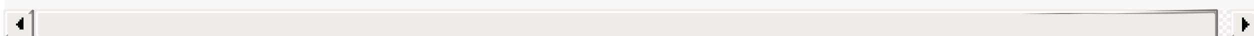
```
# Convolutional Layer 1.
filter_size1 = 5          # Convolution filters are 5 x 5 pixels.

num_filters1 = 16         # There are 16 of these filters.

# Convolutional Layer 2.
filter_size2 = 5          # Convolution filters are 5 x 5 pixels.

num_filters2 = 36         # There are 36 of these filters.

# Fully-connected layer.
fc_size = 128             # Number of neurons in fully-connected
 layer.
```



载入数据

MNIST数据集大约12MB，如果没在文件夹中找到就会自动下载。

```
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets('data/MNIST/', one_hot=True)
```

```
Extracting data/MNIST/train-images-idx3-ubyte.gz
Extracting data/MNIST/train-labels-idx1-ubyte.gz
Extracting data/MNIST/t10k-images-idx3-ubyte.gz
Extracting data/MNIST/t10k-labels-idx1-ubyte.gz
```

现在已经载入了MNIST数据集，它由70,000张图像和对应的标签（比如图像的类别）组成。数据集分成三份互相独立的子集。我们在教程中只用训练集和测试集。

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(data.train.labels)))
print("- Test-set:\t\t{}".format(len(data.test.labels)))
print("- Validation-set:\t{}".format(len(data.validation.labels)
))
```

```
Size of:
- Training-set:		55000
- Test-set:		10000
- Validation-set:	5000
```

类型标签使用One-Hot编码，这意外每个标签是长为10的向量，除了一个元素之外，其他的都为零。这个元素的索引就是类别的数字，即相应图片中画的数字。我们也需要测试数据集类别数字的整型值，用下面的方法来计算。

```
data.test.cls = np.argmax(data.test.labels, axis=1)
```

数据维度

在下面的源码中，有很多地方用到了数据维度。它们只在一个地方定义，因此我们可以在代码中使用这些数字而不是直接写数字。

```
# We know that MNIST images are 28 pixels in each dimension.
img_size = 28

# Images are stored in one-dimensional arrays of this length.
img_size_flat = img_size * img_size

# Tuple with height and width of images used to reshape arrays.
img_shape = (img_size, img_size)

# Number of colour channels for the images: 1 channel for gray-s
cale.
num_channels = 1

# Number of classes, one class for each of 10 digits.
num_classes = 10
```

用来绘制图片的帮助函数

这个函数用来在3x3的栅格中画9张图像，然后在每张图像下面写出真实类别和预测类别。

```
def plot_images(images, cls_true, cls_pred=None):
    assert len(images) == len(cls_true) == 9

    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_shape), cmap='binary')

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i],
cls_pred[i])

        # Show the classes as the label on the x-axis.
        ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

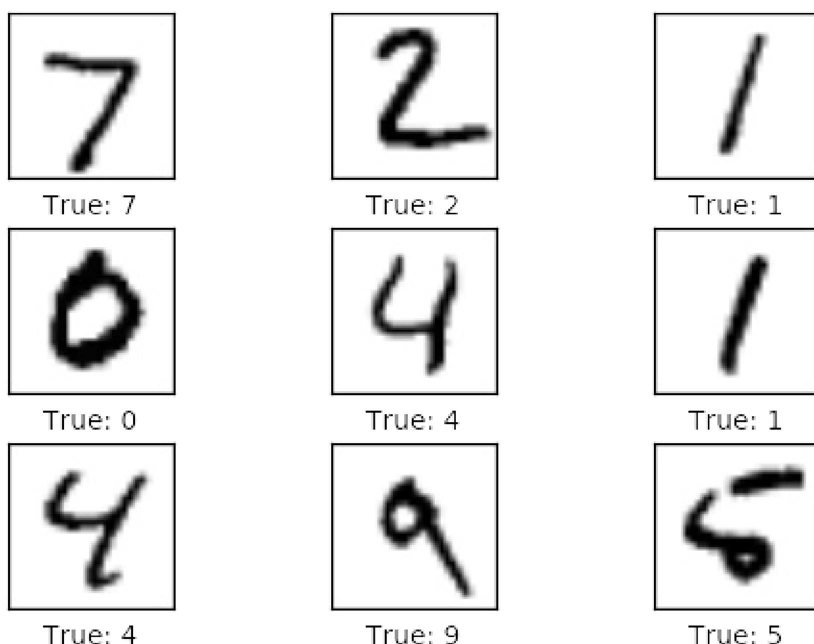
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

绘制几张图像来看看数据是否正确

```
# Get the first images from the test-set.
images = data.test.images[0:9]

# Get the true classes for those images.
cls_true = data.test.cls[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true)
```



TensorFlow图

TensorFlow的全部目的就是使用一个称之为计算图（computational graph）的东西，它会比直接在Python中进行相同计算量要高效得多。TensorFlow比Numpy更高效，因为TensorFlow了解整个需要运行的计算图，然而Numpy只知道某个时间点上唯一的数学运算。

TensorFlow也能够自动地计算需要优化的变量的梯度，使得模型有更好的表现。这是由于图是简单数学表达式的结合，因此整个图的梯度可以用链式法则推导出来。

TensorFlow还能利用多核CPU和GPU，Google也为TensorFlow制造了称为TPUs（Tensor Processing Units）的特殊芯片，它比GPU更快。

一个TensorFlow图由下面几个部分组成，后面会详细描述：

- 占位符变量（Placeholder）用来改变图的输入。
- 模型变量（Model）将会被优化，使得模型表现得更好。
- 模型本质上就是一些数学函数，它根据Placeholder和模型的输入变量来计算一

些输出。
- 一个cost度量用来指导变量的优化。
- 一个优化策略会更新模型的变量。

另外，TensorFlow图也包含了一些调试状态，比如用TensorBoard打印log数据，本教程不涉及这些。

Helper-functions for creating new variables

创建新变量的帮助函数

函数用来根据给定大小创建TensorFlow变量，并将它们用随机值初始化。需注意的是在此时并未完成初始化工作，仅仅是在TensorFlow图里定义它们。

```
def new_weights(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
```

```
def new_biases(length):
    return tf.Variable(tf.constant(0.05, shape=[length]))
```

创建卷积层的帮助函数

这个函数为TensorFlow在计算图里创建了新的卷积层。这里并没有执行什么计算，只是在TensorFlow图里添加了数学公式。

假设输入的是四维的张量，各个维度如下：

1. 图像数量
2. 每张图像的Y轴
3. 每张图像的X轴
4. 每张图像的通道数

输入通道可能是彩色通道，当输入是前面的卷积层生成的时候，它也可能是滤波通道。

输出是另外一个4通道的张量，如下：

1. 图像数量，与输入相同
2. 每张图像的Y轴。如果用到了2x2的池化，是输入图像宽高的一半。
3. 每张图像的X轴。同上。
4. 卷积滤波生成的通道数。

```
def new_conv_layer(input,              # The previous layer.
                   num_input_channels, # Num. channels in prev.
layer.
                   filter_size,        # Width and height of eac
```

```
h filter.
                   num_filters,        # Number of filters.
                   use_pooling=True):  # Use 2x2 max-pooling.

    # Shape of the filter-weights for the convolution.
    # This format is determined by the TensorFlow API.
    shape = [filter_size, filter_size, num_input_channels, num_f
ilters]

    # Create new weights aka. filters with the given shape.
    weights = new_weights(shape=shape)

    # Create new biases, one for each filter.
    biases = new_biases(length=num_filters)

    # Create the TensorFlow operation for convolution.
    # Note the strides are set to 1 in all dimensions.
    # The first and last stride must always be 1,
    # because the first is for the image-number and
    # the last is for the input-channel.
    # But e.g. strides=[1, 2, 2, 1] would mean that the filter
    # is moved 2 pixels across the x- and y-axis of the image.
    # The padding is set to 'SAME' which means the input image
    # is padded with zeroes so the size of the output is the sam
e.
    layer = tf.nn.conv2d(input=input,
                         filter=weights,
                         strides=[1, 1, 1, 1],
                         padding='SAME')

    # Add the biases to the results of the convolution.
    # A bias-value is added to each filter-channel.
    layer += biases

    # Use pooling to down-sample the image resolution?
    if use_pooling:
        # This is 2x2 max-pooling, which means that we
        # consider 2x2 windows and select the largest value
        # in each window. Then we move 2 pixels to the next wind
ow.
        layer = tf.nn.max_pool(value=layer,
                               ksize=[1, 2, 2, 1],
                               strides=[1, 2, 2, 1],
                               padding='SAME')

    # Rectified Linear Unit (ReLU).
    # It calculates max(x, 0) for each input pixel x.
    # This adds some non-linearity to the formula and allows us
    # to learn more complicated functions.
    layer = tf.nn.relu(layer)

    # Note that ReLU is normally executed before the pooling,
    # but since relu(max_pool(x)) == max_pool(relu(x)) we can
```

```
    # save 75% of the relu-operations by max-pooling first.

    # We return both the resulting layer and the filter-weights
    # because we will plot the weights later.
    return layer, weights
```

转换一个层的帮助函数

卷积层生成了4维的张量。我们会在卷积层之后添加一个全连接层，因此我们需要将这个4维的张量转换成可被全连接层使用的2维张量。

```
def flatten_layer(layer):
    # Get the shape of the input layer.
    layer_shape = layer.get_shape()

    # The shape of the input layer is assumed to be:
    # layer_shape == [num_images, img_height, img_width, num_cha
nnels]

    # The number of features is: img_height * img_width * num_ch
annels
    # We can use a function from TensorFlow to calculate this.
    num_features = layer_shape[1:4].num_elements()

    # Reshape the layer to [num_images, num_features].
    # Note that we just set the size of the second dimension
    # to num_features and the size of the first dimension to -1
    # which means the size in that dimension is calculated
    # so the total size of the tensor is unchanged from the resh
aping.
    layer_flat = tf.reshape(layer, [-1, num_features])

    # The shape of the flattened layer is now:
    # [num_images, img_height * img_width * num_channels]

    # Return both the flattened layer and the number of features.

    return layer_flat, num_features
```

创建一个全连接层的帮助函数

这个函数为TensorFlow在计算图中创建了一个全连接层。这里也不进行任何计算，只是往TensorFlow图中添加数学公式。

输入是大小为 `[num_images, num_inputs]` 的二维张量。输出是大小为 `[num_images, num_outputs]` 的2维张量。

```
def new_fc_layer(input,          # The previous layer.
                 num_inputs,     # Num. inputs from prev. layer.
                 num_outputs,    # Num. outputs.
                 use_relu=True): # Use Rectified Linear Unit (Re
LU)?

    # Create new weights and biases.
    weights = new_weights(shape=[num_inputs, num_outputs])
    biases = new_biases(length=num_outputs)

    # Calculate the layer as the matrix multiplication of
    # the input and weights, and then add the bias-values.
    layer = tf.matmul(input, weights) + biases

    # Use ReLU?
    if use_relu:
        layer = tf.nn.relu(layer)

    return layer
```

占位符（Placeholder）变量

Placeholder是作为图的输入，每次我们运行图的时候都可能会改变它们。将这个过程称为feeding placeholder变量，后面将会描述它。

首先我们为输入图像定义placeholder变量。这让我们可以改变输入到TensorFlow图中的图像。这也是一个张量（tensor），代表一个多维向量或矩阵。数据类型设置为float32，形状设为 `[None, img_size_flat]` ， `None` 代表tensor可能保存着任意数量的图像，每张图象是一个长度为 `img_size_flat` 的向量。

```
x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name=
'x')
```

卷积层希望 `x` 被编码为4维张量，因此我们需要将它的形状转换至 `[num_images, img_height, img_width, num_channels]` 。注意 `img_height == img_width == img_size` ，如果第一维的大小设为-1， `num_images` 的大小也会被自动推导出来。转换运算如下：

```
x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
```

接下来我们为输入变量 `x` 中的图像所对应的真实标签定义placeholder变量。变量的形状是 `[None, num_classes]` ，这代表着它保存了任意数量的标签，每个标签是长度为 `num_classes` 的向量，本例中长度为10。

```
y_true = tf.placeholder(tf.float32, shape=[None, 10], name='y_tr
ue')
```

我们也可以为class-number提供一个placeholder，但这里用argmax来计算它。这里只是TensorFlow中的一些操作，没有执行什么运算。

```
y_true_cls = tf.argmax(y_true, dimension=1)
```

卷积层 1

创建第一个卷积层。将 `x_image` 当作输入，创建 `num_filters1` 个不同的滤波器，每个滤波器的宽高都与 `filter_size1` 相等。最终我们会用2x2的max-pooling将图像降采样，使它的尺寸减半。

```
layer_conv1, weights_conv1 = \
    new_conv_layer(input=x_image,
                   num_input_channels=num_channels,
                   filter_size=filter_size1,
                   num_filters=num_filters1,
                   use_pooling=True)
```

检查卷积层输出张量的大小。它是（？,14, 14, 16），这代表着有任意数量的图像（？代表数量），每张图像有14个像素的宽和高，有16个不同的通道，每个滤波器各有一个通道。

```
layer_conv1
```

```
<tf.Tensor 'Relu:0' shape=(?, 14, 14, 16) dtype=float32>
```

卷积层 2

创建第二个卷积层，它将第一个卷积层的输出作为输入。输入通道的数量对应着第一个卷积层的滤波数。

```
layer_conv2, weights_conv2 = \
    new_conv_layer(input=layer_conv1,
                   num_input_channels=num_filters1,
                   filter_size=filter_size2,
                   num_filters=num_filters2,
                   use_pooling=True)
```

核对一下这个卷积层输出张量的大小。它的大小是（？，7，7，36）,其中？也代表着任意数量的图像，每张图有7像素的宽高，每个滤波器有36个通道。

```
layer_conv2
```

```
<tf.Tensor 'Relu_1:0' shape=(?, 7, 7, 36) dtype=float32>
```

转换层

这个卷积层输出一个4维张量。现在我们想将它作为一个全连接网络的输入，这就需要将它转换成2维张量。

```
layer_flat, num_features = flatten_layer(layer_conv2)
```

这个张量的大小是（？，1764），意味着共有一定数量的图像，每张图像被转换成长为1764的向量。其中1764 = 7 x 7 x 36。

```
layer_flat
```

```
<tf.Tensor 'Reshape_1:0' shape=(?, 1764) dtype=float32>
```

```
num_features
```

```
1764
```

全连接层 1

往网络中添加一个全连接层。输入是一个前面卷积得到的被转换过的层。全连接层中的神经元或节点数为 `fc_size` 。我们可以用ReLU来学习非线性关系。

```
layer_fc1 = new_fc_layer(input=layer_flat,
                         num_inputs=num_features,
                         num_outputs=fc_size,
                         use_relu=True)
```

全连接层的输出是一个大小为（？，128）的张量，？代表着一定数量的图像，并且 `fc_size` == 128。

```
layer_fc1
```

```
<tf.Tensor 'Relu_2:0' shape=(?, 128) dtype=float32>
```

全连接层 2

添加另外一个全连接层，它的输出是一个长度为10的向量，它确定了输入图是属于哪个类别。这层并没有用到ReLU。

```
layer_fc2 = new_fc_layer(input=layer_fc1,
                         num_inputs=fc_size,
                         num_outputs=num_classes,
                         use_relu=False)
```

```
layer_fc2
```

```
<tf.Tensor 'add_3:0' shape=(?, 10) dtype=float32>
```

预测类别

第二个全连接层估算了输入图有多大的可能属于10个类别中的其中一个。然而，这是很粗略的估计并且很难解释，因为数值可能很小或很大，因此我们会对它们做归一化，将每个元素限制在0到1之间，并且相加为1。这用一个称为softmax的函数来计算的，结果保存在 `y_pred` 中。

```
y_pred = tf.nn.softmax(layer_fc2)
```

类别数字是最大元素的索引。

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

优化损失函数

为了使模型更好地对输入图像进行分类，我们必须改变 `weights` 和 `biases` 变量。首先我们需要对比模型 `y_pred` 的预测输出和期望输出的 `y_true` ，来了解目前模型的性能如何。

交叉熵（cross-entropy）是在分类中使用的性能度量。交叉熵是一个常为正值的连续函数，如果模型的预测值精准地符合期望的输出，它就等于零。因此，优化的目的就是通过改变网络层的变量来最小化交叉熵。

TensorFlow有一个内置的计算交叉熵的函数。这个函数内部计算了softmax，所以我们要用 `layer_fc2` 的输出而非直接用 `y_pred` ,因为 `y_pred` 上已经计算了softmax。

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=l
ayer_fc2,
                                                         labels=y
_true)
```

我们为每个图像分类计算了交叉熵，所以有一个当前模型在每张图上表现的度量。但是为了用交叉熵来指导模型变量的优化，我们需要一个额外的标量值，因此简单地利用所有图像分类交叉熵的均值。

```
cost = tf.reduce_mean(cross_entropy)
```

优化方法

既然我们有一个需要被最小化的损失度量，接着就可以建立优化一个优化器。这个例子中，我们使用的是梯度下降的变体 `AdamOptimizer` 。

优化过程并不是在这里执行。实际上，还没计算任何东西，我们只是往TensorFlow图中添加了优化器，以便之后的操作。

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(
cost)
```

性能度量

我们需要另外一些性能度量，来向用户展示这个过程。

这是一个布尔值向量，代表预测类型是否等于每张图片的真实类型。

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

上面的计算先将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确度。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
))
```

运行TensorFlow

创建TensorFlow会话（session）

一旦创建了TensorFlow图，我们需要创建一个TensorFlow会话，用来运行图。

```
session = tf.Session()
```

初始化变量

我们需要在开始优化weights和biases变量之前对它们进行初始化。

```
session.run(tf.global_variables_initializer())
```

用来优化迭代的帮助函数

在训练集中有50,000张图。用这些图像计算模型的梯度会花很多时间。因此我们利用随机梯度下降的方法，它在优化器的每次迭代里只用到了一小部分的图像。

如果内存耗尽导致电脑死机或变得很慢，你应该试着减少这些数量，但同时可能还需要更优化的迭代。

```
train_batch_size = 64
```

函数执行了多次的优化迭代来逐步地提升网络层的变量。在每次迭代中，从训练集中选择一批新的数据，然后TensorFlow用这些训练样本来执行优化器。每100次迭代会打印出相关信息。

```
# Counter for total number of iterations performed so far.
total_iterations = 0

def optimize(num_iterations):
    # Ensure we update the global variable rather than a local c
opy.
    global total_iterations

    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(total_iterations,
                   total_iterations + num_iterations):

        # Get a batch of training examples.
        # x_batch now holds a batch of images and
```

```
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = data.train.next_batch(train_batc
h_size)

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimiz
er.
        session.run(optimizer, feed_dict=feed_dict_train)

        # Print status every 100 iterations.
        if i % 100 == 0:
            # Calculate the accuracy on the training-set.
            acc = session.run(accuracy, feed_dict=feed_dict_trai
n)

            # Message for printing.
            msg = "Optimization Iteration: {0:>6}, Training Accu
racy: {1:>6.1%}"

            # Print it.
            print(msg.format(i + 1, acc))

    # Update the total number of iterations performed.
    total_iterations += num_iterations

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_
dif)))))
```

用来绘制错误样本的帮助函数

函数用来绘制测试集中被误分类的样本。

```
def plot_example_errors(cls_pred, correct):
    # This function is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # correct is a boolean array whether the predicted class
    # is equal to the true class for each image in the test-set.

    # Negate the boolean array.
    incorrect = (correct == False)

    # Get the images from the test-set that have been
    # incorrectly classified.
    images = data.test.images[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = data.test.cls[incorrect]

    # Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])
```

绘制混淆（confusion）矩阵的帮助函数

```
def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the true classifications for the test-set.
    cls_true = data.test.cls

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_true,
                          y_pred=cls_pred)

    # Print the confusion matrix as text.
    print(cm)

    # Plot the confusion matrix as an image.
    plt.matshow(cm)

    # Make various adjustments to the plot.
    plt.colorbar()
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
    plt.yticks(tick_marks, range(num_classes))
    plt.xlabel('Predicted')
    plt.ylabel('True')

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

展示性能的帮助函数

函数用来打印测试集上的分类准确度。

为测试集上的所有图片计算分类会花费一段时间，因此我们直接用这个函数来调用上面的结果，这样就不用每次都重新计算了。

这个函数可能会占用很多电脑内存，这也是为什么将测试集分成更小的几个部分。如果你的电脑内存比较小或死机了，就要试着降低batch-size。

```
# Split the test-set into smaller batches of this size.
test_batch_size = 256

def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # Number of images in the test-set.
    num_test = len(data.test.images)
```

```
    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_test, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing thi
s.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_test:
        # The ending index for the next batch is denoted j.
        j = min(i + test_batch_size, num_test)

        # Get the images from the test-set between index i and j.

        images = data.test.images[i:j, :]

        # Get the associated labels.
        labels = data.test.labels[i:j, :]

        # Create a feed-dict with these images and labels.
        feed_dict = {x: images,
                     y_true: labels}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_d
ict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Convenience variable for the true class-numbers of the tes
t-set.
    cls_true = data.test.cls

    # Create a boolean array whether each image is correctly cla
ssified.
    correct = (cls_true == cls_pred)

    # Calculate the number of correctly classified images.
    # When summing a boolean array, False means 0 and True means
 1.
    correct_sum = correct.sum()

    # Classification accuracy is the number of correctly classif
ied
    # images divided by the total number of images in the test-s
et.
    acc = float(correct_sum) / num_test
```

```
    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, correct_sum, num_test))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

优化之前的性能

测试集上的准确度很低，这是由于模型只做了初始化，并没做任何优化，所以它只是对图像做随机分类。

```
print_test_accuracy()
```

```
Accuracy on Test-Set: 10.9% (1093 / 10000)
```

1次迭代后的性能

做了一次优化后，此时优化器的学习率很低，性能其实并没有多大提升。

```
optimize(num_iterations=1)
```

```
Optimization Iteration:      1, Training Accuracy:   6.2%
Time usage: 0:00:00
```

```
print_test_accuracy()
```

```
Accuracy on Test-Set: 13.0% (1296 / 10000)
```

100次迭代优化后的性能

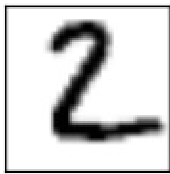
100次优化迭代之后，模型显著地提升了分类的准确度。

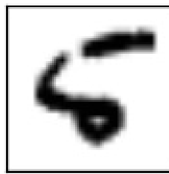
```
optimize(num_iterations=99) # We already performed 1 iteration above.
```

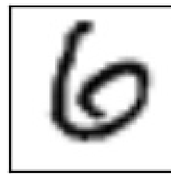
```
Time usage: 0:00:00
```

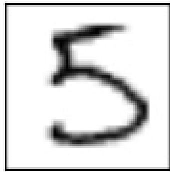
```
print_test_accuracy(show_example_errors=True)
```

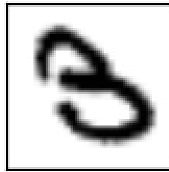
```
Accuracy on Test-Set: 66.6% (6656 / 10000)
Example errors:
```

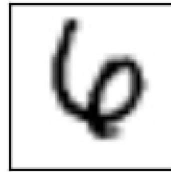

True: 2, Pred: 1

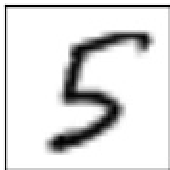

True: 5, Pred: 4

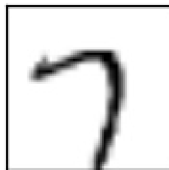

True: 6, Pred: 0


True: 5, Pred: 3


True: 3, Pred: 6


True: 6, Pred: 4


True: 5, Pred: 6

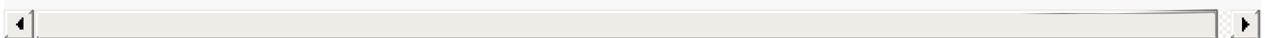

True: 7, Pred: 9


True: 4, Pred: 0

1000次优化迭代后的性能

1000次优化迭代之后，模型在测试集上的准确度超过了90%。

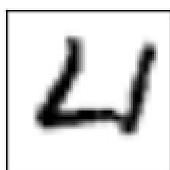
```
optimize(num_iterations=900) # We performed 100 iterations above.
```

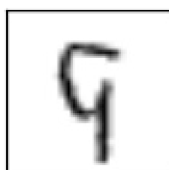


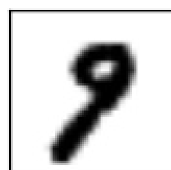
```
Optimization Iteration:    101, Training Accuracy:  71.9%
Optimization Iteration:    201, Training Accuracy:  76.6%
Optimization Iteration:    301, Training Accuracy:  71.9%
Optimization Iteration:    401, Training Accuracy:  85.9%
Optimization Iteration:    501, Training Accuracy:  89.1%
Optimization Iteration:    601, Training Accuracy:  95.3%
Optimization Iteration:    701, Training Accuracy:  90.6%
Optimization Iteration:    801, Training Accuracy:  92.2%
Optimization Iteration:    901, Training Accuracy:  95.3%
Time usage: 0:00:03
```

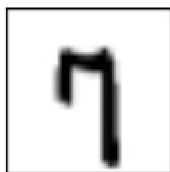
```
print_test_accuracy(show_example_errors=True)
```

```
Accuracy on Test-Set: 93.1% (9308 / 10000)
Example errors:
```

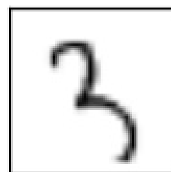

True: 4, Pred: 6

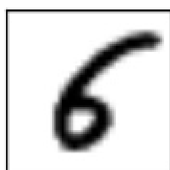

True: 9, Pred: 4

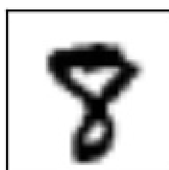

True: 9, Pred: 7

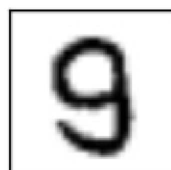

True: 7, Pred: 9


True: 5, Pred: 4


True: 3, Pred: 1


True: 6, Pred: 5


True: 8, Pred: 7


True: 9, Pred: 8

10,000次优化迭代后的性能

经过10,000次优化迭代后，测试集上的分类准确率高达99%。

```
optimize(num_iterations=9000) # We performed 1000 iterations above.
```

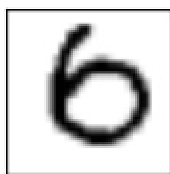
```
Optimization Iteration:   1001, Training Accuracy:  98.4%
```

```
Optimization Iteration:   1101, Training Accuracy:  93.8%
Optimization Iteration:   1201, Training Accuracy:  92.2%
Optimization Iteration:   1301, Training Accuracy:  95.3%
Optimization Iteration:   1401, Training Accuracy:  93.8%
Optimization Iteration:   1501, Training Accuracy:  93.8%
Optimization Iteration:   1601, Training Accuracy:  92.2%
Optimization Iteration:   1701, Training Accuracy:  92.2%
Optimization Iteration:   1801, Training Accuracy:  89.1%
Optimization Iteration:   1901, Training Accuracy:  95.3%
Optimization Iteration:   2001, Training Accuracy:  93.8%
Optimization Iteration:   2101, Training Accuracy:  98.4%
Optimization Iteration:   2201, Training Accuracy:  92.2%
Optimization Iteration:   2301, Training Accuracy:  95.3%
Optimization Iteration:   2401, Training Accuracy: 100.0%
Optimization Iteration:   2501, Training Accuracy:  96.9%
Optimization Iteration:   2601, Training Accuracy:  93.8%
Optimization Iteration:   2701, Training Accuracy: 100.0%
Optimization Iteration:   2801, Training Accuracy:  95.3%
Optimization Iteration:   2901, Training Accuracy:  95.3%
Optimization Iteration:   3001, Training Accuracy:  96.9%
Optimization Iteration:   3101, Training Accuracy:  96.9%
Optimization Iteration:   3201, Training Accuracy:  95.3%
Optimization Iteration:   3301, Training Accuracy:  96.9%
Optimization Iteration:   3401, Training Accuracy:  98.4%
Optimization Iteration:   3501, Training Accuracy: 100.0%
Optimization Iteration:   3601, Training Accuracy:  98.4%
Optimization Iteration:   3701, Training Accuracy:  95.3%
Optimization Iteration:   3801, Training Accuracy:  95.3%
Optimization Iteration:   3901, Training Accuracy:  95.3%
Optimization Iteration:   4001, Training Accuracy: 100.0%
Optimization Iteration:   4101, Training Accuracy:  93.8%
Optimization Iteration:   4201, Training Accuracy:  95.3%
Optimization Iteration:   4301, Training Accuracy: 100.0%
Optimization Iteration:   4401, Training Accuracy:  96.9%
Optimization Iteration:   4501, Training Accuracy: 100.0%
Optimization Iteration:   4601, Training Accuracy: 100.0%
Optimization Iteration:   4701, Training Accuracy: 100.0%
Optimization Iteration:   4801, Training Accuracy:  98.4%
Optimization Iteration:   4901, Training Accuracy:  98.4%
Optimization Iteration:   5001, Training Accuracy:  98.4%
Optimization Iteration:   5101, Training Accuracy: 100.0%
Optimization Iteration:   5201, Training Accuracy:  95.3%
Optimization Iteration:   5301, Training Accuracy:  96.9%
Optimization Iteration:   5401, Training Accuracy: 100.0%
Optimization Iteration:   5501, Training Accuracy: 100.0%
Optimization Iteration:   5601, Training Accuracy: 100.0%
Optimization Iteration:   5701, Training Accuracy:  96.9%
Optimization Iteration:   5801, Training Accuracy:  98.4%
Optimization Iteration:   5901, Training Accuracy: 100.0%
Optimization Iteration:   6001, Training Accuracy:  95.3%
Optimization Iteration:   6101, Training Accuracy:  96.9%
Optimization Iteration:   6201, Training Accuracy: 100.0%
Optimization Iteration:   6301, Training Accuracy:  96.9%
```

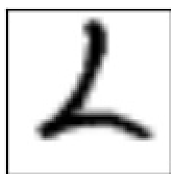
```
Optimization Iteration:   6401, Training Accuracy: 100.0%
Optimization Iteration:   6501, Training Accuracy:  98.4%
Optimization Iteration:   6601, Training Accuracy:  98.4%
Optimization Iteration:   6701, Training Accuracy:  95.3%
Optimization Iteration:   6801, Training Accuracy: 100.0%
Optimization Iteration:   6901, Training Accuracy:  98.4%
Optimization Iteration:   7001, Training Accuracy:  95.3%
Optimization Iteration:   7101, Training Accuracy: 100.0%
Optimization Iteration:   7201, Training Accuracy: 100.0%
Optimization Iteration:   7301, Training Accuracy: 100.0%
Optimization Iteration:   7401, Training Accuracy: 100.0%
Optimization Iteration:   7501, Training Accuracy: 100.0%
Optimization Iteration:   7601, Training Accuracy:  96.9%
Optimization Iteration:   7701, Training Accuracy:  98.4%
Optimization Iteration:   7801, Training Accuracy:  95.3%
Optimization Iteration:   7901, Training Accuracy: 100.0%
Optimization Iteration:   8001, Training Accuracy: 100.0%
Optimization Iteration:   8101, Training Accuracy:  98.4%
Optimization Iteration:   8201, Training Accuracy:  98.4%
Optimization Iteration:   8301, Training Accuracy: 100.0%
Optimization Iteration:   8401, Training Accuracy:  96.9%
Optimization Iteration:   8501, Training Accuracy:  98.4%
Optimization Iteration:   8601, Training Accuracy:  98.4%
Optimization Iteration:   8701, Training Accuracy: 100.0%
Optimization Iteration:   8801, Training Accuracy: 100.0%
Optimization Iteration:   8901, Training Accuracy:  98.4%
Optimization Iteration:   9001, Training Accuracy:  95.3%
Optimization Iteration:   9101, Training Accuracy: 100.0%
Optimization Iteration:   9201, Training Accuracy: 100.0%
Optimization Iteration:   9301, Training Accuracy:  96.9%
Optimization Iteration:   9401, Training Accuracy:  96.9%
Optimization Iteration:   9501, Training Accuracy:  98.4%
Optimization Iteration:   9601, Training Accuracy: 100.0%
Optimization Iteration:   9701, Training Accuracy:  96.9%
Optimization Iteration:   9801, Training Accuracy:  98.4%
Optimization Iteration:   9901, Training Accuracy:  98.4%
Time usage: 0:00:26
```

```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

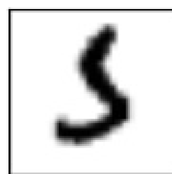
```
Accuracy on Test-Set: 98.8% (9880 / 10000)
Example errors:
```



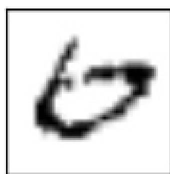
True: 6, Pred: 0



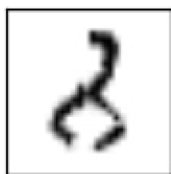
True: 2, Pred: 1



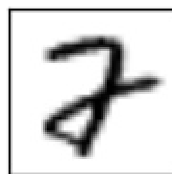
True: 5, Pred: 3



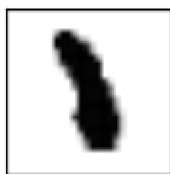
True: 6, Pred: 0



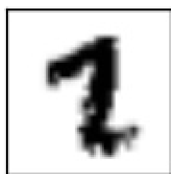
True: 8, Pred: 2



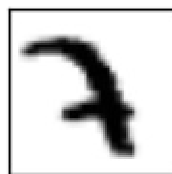
True: 2, Pred: 7



True: 1, Pred: 8

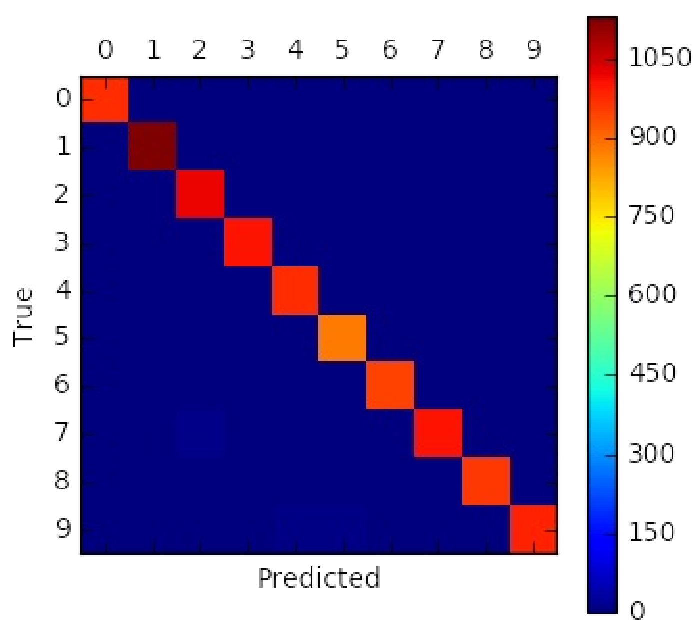


True: 2, Pred: 1



True: 7, Pred: 3

```
Confusion Matrix:
[[ 973    0    1    0    0    1    1    0    3    1]
 [    0 1129    2    1    0    0    1    1    1    0]
 [    1    2 1023    2    0    0    0    2    2    0]
 [    1    0    1 1002    0    3    0    1    2    0]
 [    0    1    0    0  974    0    1    0    2    4]
 [    2    0    0    3    0  882    2    0    1    2]
 [    4    1    0    0    1    4  948    0    0    0]
 [    1    4   11    2    0    0    0 1004    2    4]
 [    3    0    4    2    1    2    0    0  960    2]
 [    3    4    1    0    7    5    0    2    2  985]]
```



权重和层的可视化

为了理解为什么卷积神经网络可以识别手写数字，我们将会对卷积滤波和输出图像进行可视化。

绘制卷积权重的帮助函数

```
def plot_conv_weights(weights, input_channel=0):
    # Assume weights are TensorFlow ops for 4-dim variables
    # e.g. weights_conv1 or weights_conv2.

    # Retrieve the values of the weight-variables from TensorFlo
w.
    # A feed-dict is not necessary because nothing is calculated.

    w = session.run(weights)

    # Get the lowest and highest values for the weights.
    # This is used to correct the colour intensity across
    # the images so they can be compared with each other.
    w_min = np.min(w)
    w_max = np.max(w)

    # Number of filters used in the conv. layer.
    num_filters = w.shape[3]

    # Number of grids to plot.
    # Rounded-up, square-root of the number of filters.
    num_grids = math.ceil(math.sqrt(num_filters))

    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(num_grids, num_grids)

    # Plot all the filter-weights.
    for i, ax in enumerate(axes.flat):
        # Only plot the valid filter-weights.
        if i<num_filters:
            # Get the weights for the i'th filter of the input c
hannel.
            # See new_conv_layer() for details on the format
            # of this 4-dim tensor.
            img = w[:, :, input_channel, i]

            # Plot image.
            ax.imshow(img, vmin=w_min, vmax=w_max,
                      interpolation='nearest', cmap='seismic')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

绘制卷积层输出的帮助函数

```
def plot_conv_layer(layer, image):
    # Assume layer is a TensorFlow op that outputs a 4-dim tensor

    # which is the output of a convolutional layer,
    # e.g. layer_conv1 or layer_conv2.

    # Create a feed-dict containing just one image.
    # Note that we don't need to feed y_true because it is
    # not used in this calculation.
    feed_dict = {x: [image]}

    # Calculate and retrieve the output values of the layer
    # when inputting that image.
    values = session.run(layer, feed_dict=feed_dict)

    # Number of filters used in the conv. layer.
    num_filters = values.shape[3]

    # Number of grids to plot.
    # Rounded-up, square-root of the number of filters.
    num_grids = math.ceil(math.sqrt(num_filters))

    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(num_grids, num_grids)

    # Plot the output images of all the filters.
    for i, ax in enumerate(axes.flat):
        # Only plot the images for valid filters.
        if i<num_filters:
            # Get the output image of using the i'th filter.
            # See new_conv_layer() for details on the format
            # of this 4-dim tensor.
            img = values[0, :, :, i]

            # Plot image.
            ax.imshow(img, interpolation='nearest', cmap='binary'
)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

输入图像

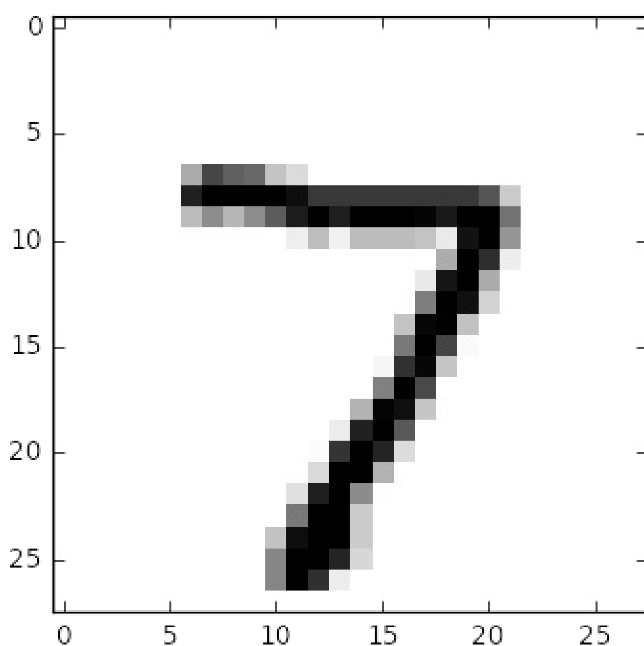
绘制图像的帮助函数

```
def plot_image(image):
    plt.imshow(image.reshape(img_shape),
               interpolation='nearest',
               cmap='binary')

    plt.show()
```

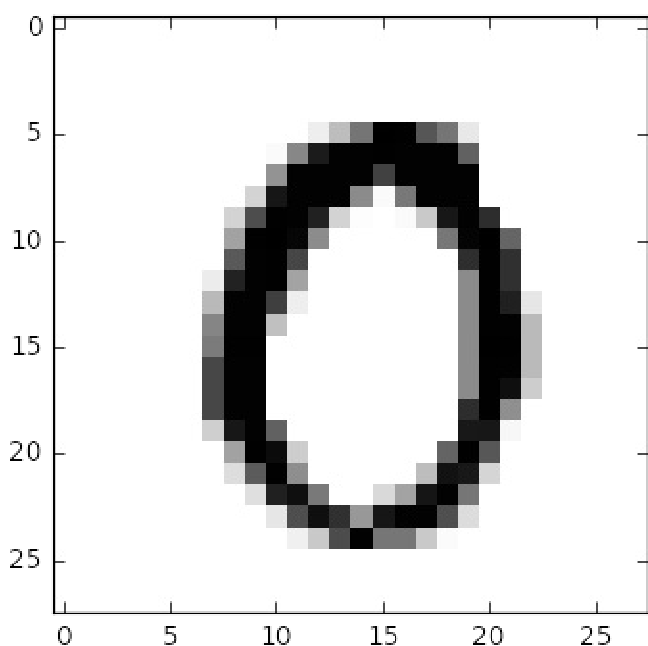
如下所示，绘制一张测试集中的图像。

```
image1 = data.test.images[0]
plot_image(image1)
```



绘制测试集里的另一张图像。

```
image2 = data.test.images[13]
plot_image(image2)
```

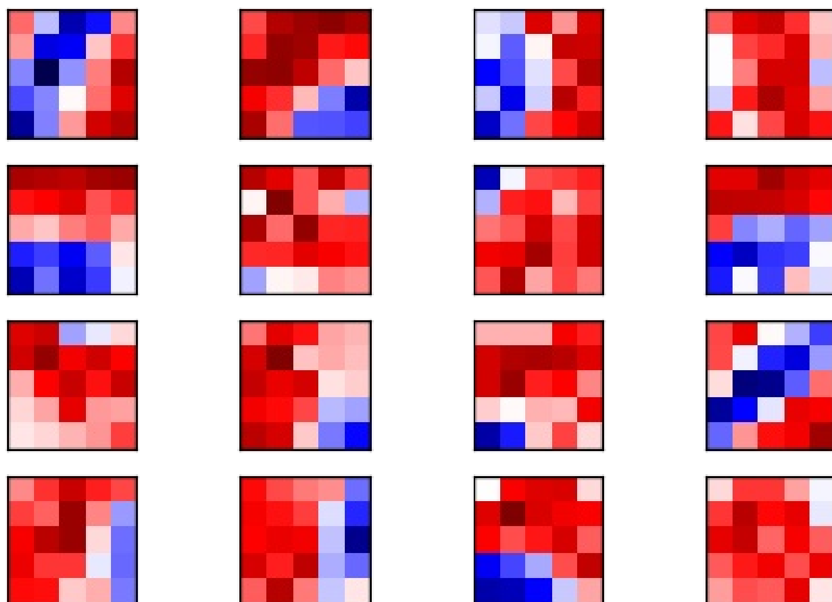


卷积层 1

现在绘制第一个卷积层的滤波权重。

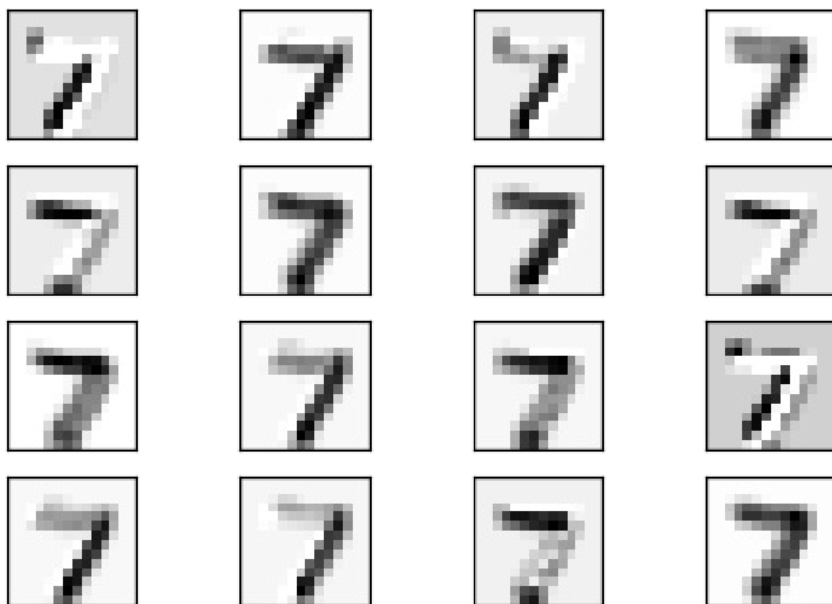
其中正值权重是红色的，负值为蓝色。

```
plot_conv_weights(weights=weights_conv1)
```



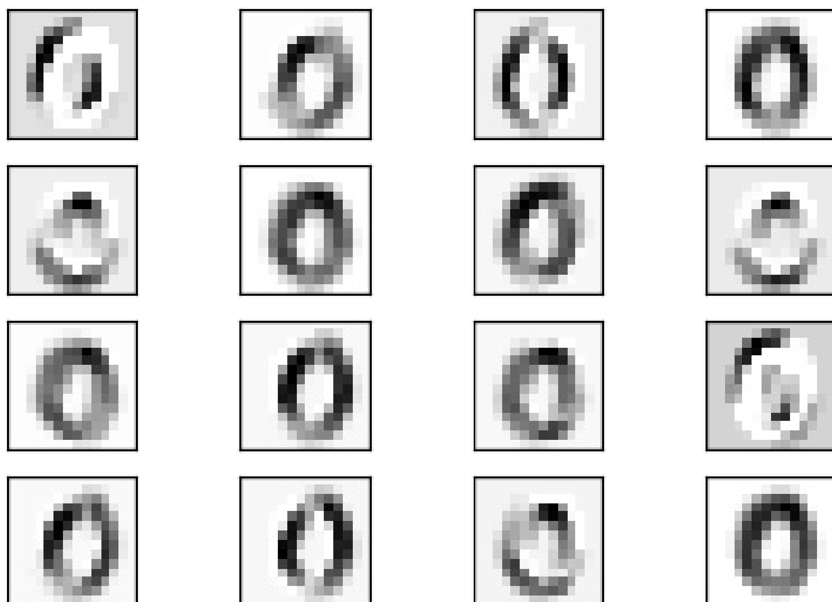
将这些卷积滤波添加到第一张输入图像，得到以下输出，它们也作为第二个卷积层的输入。注意这些图像被降采样到14 x 14像素，即原始输入图分辨率的一半。

```
plot_conv_layer(layer=layer_conv1, image=image1)
```



下面是将卷积滤波添加到第二张图像的结果。

```
plot_conv_layer(layer=layer_conv1, image=image2)
```



从这些图像很难看出卷积滤波的作用是什么。显然，它们生成了输入图像的一些变体，就像光线从不同角度打到图像上并产生阴影一样。

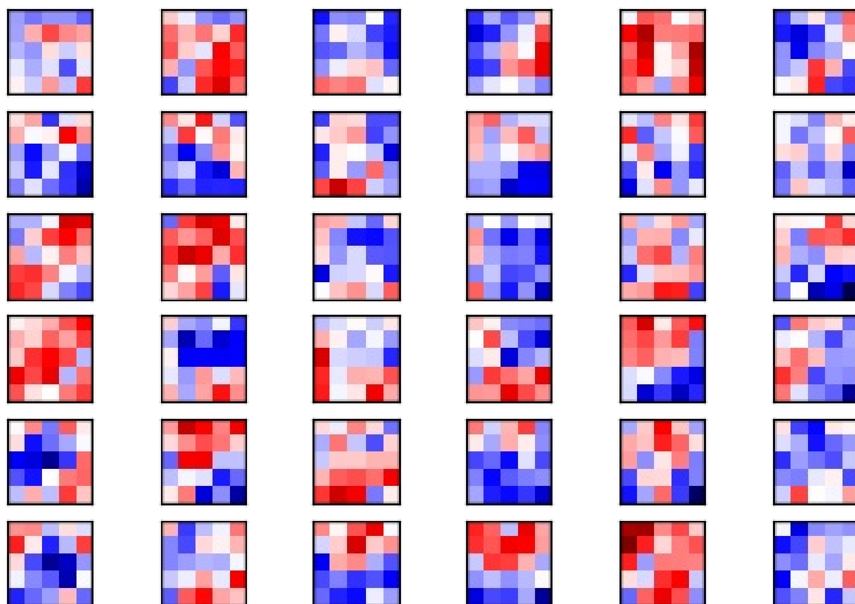
卷积层 2

现在绘制第二个卷积层的滤波权重。

第一个卷积层有16个输出通道，代表着第二个卷基层有16个输入。第二个卷积层的每个输入通道也有一些权重滤波。我们先绘制第一个通道的权重滤波。

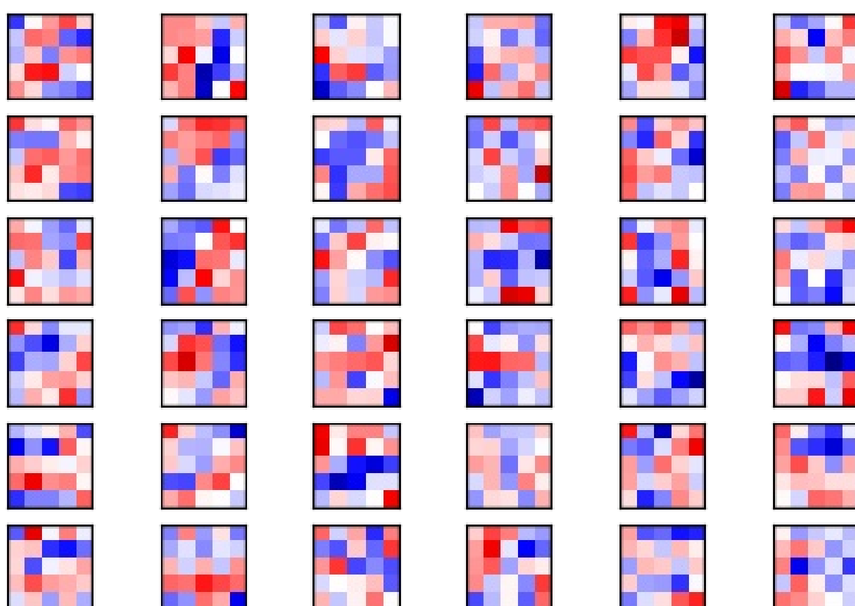
同样的，正值是红色，负值是蓝色。

```
plot_conv_weights(weights=weights_conv2, input_channel=0)
```



第二个卷积层共有16个输入通道，我们可以同样地画出其他图像。这里我们画出第二个通道的图像。

```
plot_conv_weights(weights=weights_conv2, input_channel=1)
```

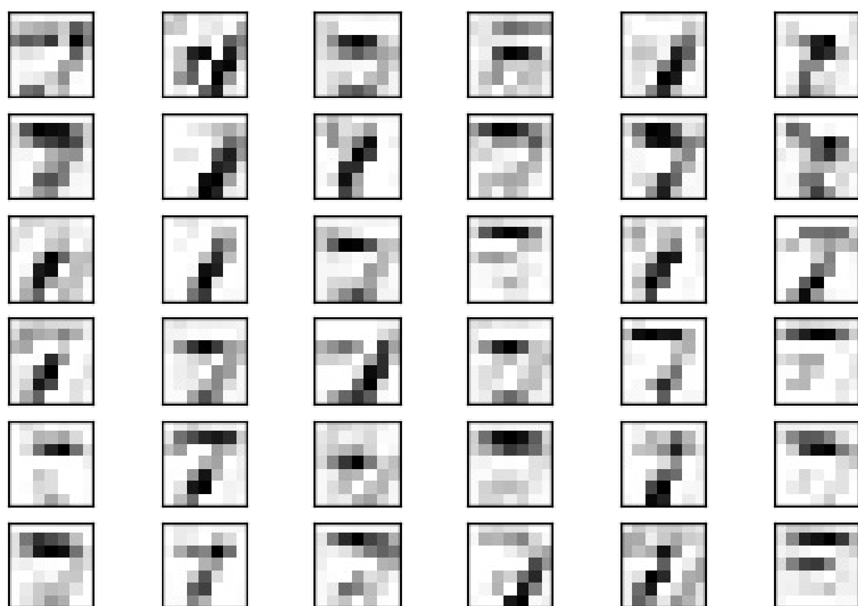


由于这些滤波是高维度的，很难理解它们是如何应用的。

给第一个卷积层的输出加上这些滤波，得到下面的图像。

这些图像被降采样至7 x 7的像素，即上一个卷积层输出的一半。

```
plot_conv_layer(layer=layer_conv2, image=image1)
```



这是给第二张图像加上滤波权重的结果。

```
plot_conv_layer(layer=layer_conv2, image=image2)
```



从这些图像来看，似乎第二个卷积层会检测输入图像中的线段和模式，这对输入图中的局部变化不那么敏感。

关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# session.close()
```

总结

我们看到卷积神经网络在识别手写数字上的表现要比教程#01中简单线性模型要好得多。卷积神经网络可能达到99%的分类准确率，如果你做一些调整，还可能表现得更好，而简单线性模型只有91%的正确率。

然而，卷积神经网络实现起来更复杂，并且光看权重滤波也不好理解为什么它能奏效或者失败。

因此我们需要一个更简单的实现卷积神经网络的方式，同时也要寻找一种更好的方法来对它们内部工作原理进行可视化。

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 如果你不改变任何参数，多次运行Notebook，会得到完成一样的结果吗？随机性的来源是什么？
- 再进行10,000次优化。结果有变好么？
- 改变优化器的学习率。
- 改变层次的属性，比如卷积滤波器数量、滤波器的大小、全连接层中的神经元数量等等。
- 在全连接层之后添加一个drop-out层。在计算分类准确率的时候，drop-out层可能为0，因此你需要一个placeholder变量。
- 改变ReLU和max-pooling的顺序。它的计算结果相同么？最快的计算方法是什么？节省了多少计算量？这也适用于Sigmoid-function和average-pooling吗？
- 添加一个或多个卷积层和全连接层。这对性能有帮助吗？
- 能得到良好结果的最小可能配置是什么？
- 试着在最后一个全连接层中使用ReLU。性能有变化吗？为什么？

- 卷积层里不用pooling。这对分类准确率和训练时间有影响吗？
- 在卷积层里用2x2的stride代替max-pooling？有什么变化吗？
- 不看源码，自己重写程序。
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #03

PrettyTensor

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

简介

之前的教程演示了如何在TensorFlow中实现一个卷积神经网络，这需要了解一些TensorFlow工作的底层原理。它有点复杂，实现起来还容易犯错。

这篇教程为我们说明了如何使用TensorFlow的一个附加包PrettyTensor，它也是Google开发的。PrettyTensor提供了在TensorFlow中创建神经网络的更简单的方法，让我们可以关注自己想要实现的想法，而不用过多担心底层的实现细节。这也让代码更短、更容易阅读和修改。

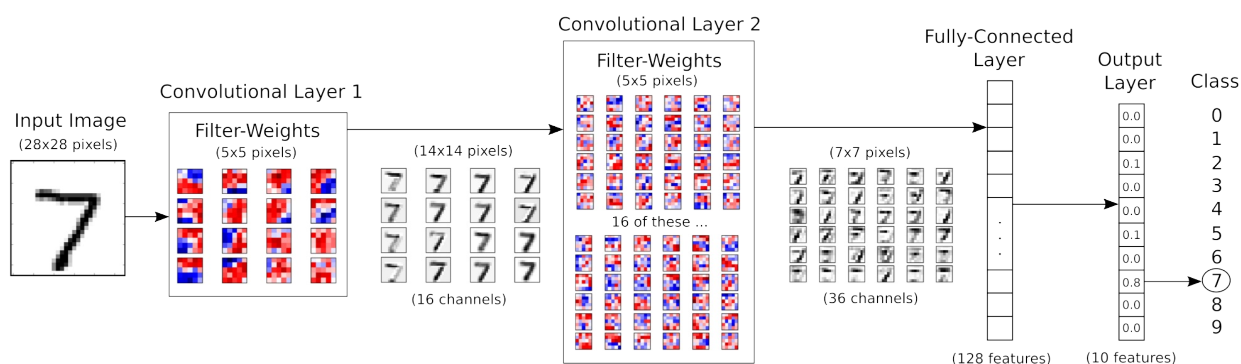
除了用PrettyTensor构造图之外，这篇教程的大部分代码和教程 #02 中的一样，当然还有一些细微的变化。

这篇教程是基于教程 #02 之上的，如果你是TensorFlow新手的话，推荐先学完上一份教程。你需要熟悉基本的线性代数、Python和Jupyter Notebook编辑器。

流程图

下面的图表直接展示了之后实现的卷积神经网络中数据的传递。关于卷积的详细描述请看上一篇教程。

```
from IPython.display import Image
Image('images/02_network_flowchart.png')
```



输入图像在第一层卷基层中用权重过滤器处理。结果在16张新图里，每个代表了卷积层里一个过滤器（的处理结果）。图像也经过降采样，因此图像分辨率从28x28减少到14x14。

这16张小图在第二个卷积层中处理。这16个通道都需要一个权重过滤，这层的输出的每个通道也各需要一个权重过滤。总共有36个输出，所以在第二个卷积层有16 x 36 = 576个滤波器。输出图再一次降采样到7x7个像素。

第二个卷积层的输出是36张7x7像素的图像。它们被压到一个长为7 x 7 x 36 = 1764的向量中去，它作为一个有128个神经元（或元素）的全连接网络的输入。这些又输入到另一个有10个神经元的全连接层中，每个神经元代表一个类别，用来确定图像的类别，也即图像上的数字。

卷积滤波一开始是随机挑选的，因此分类也是随机完成的。根据交叉熵（cross-entropy）来测量输入图预测值和真实类别间的错误。然后优化器用链式法则自动地将这个误差传在卷积网络中传递，更新滤波权重来提升分类质量。这个过程迭代了几千次，直到分类误差足够低。

这些特定的滤波权重和中间图像是一个优化的结果，和你执行这些代码所看到的可能会有所不同。

注意，这些在TensorFlow上的计算是在一部分图像上执行，而非单独的一张图，这使得计算更有效。也意味着在TensorFlow上实现时，这个流程图实际上会有更多的数据维度。

导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
import time
from datetime import timedelta
import math

# We also need PrettyTensor.
import prettytensor as pt
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.12.0-rc0'
```

PrettyTensor 版本:

```
pt.__version__
```

```
'0.7.1'
```

载入数据

MNIST数据集大约12MB，如果没在给定路径中找到就会自动下载。

```
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets('data/MNIST/', one_hot=True)
```

```
Extracting data/MNIST/train-images-idx3-ubyte.gz
Extracting data/MNIST/train-labels-idx1-ubyte.gz
Extracting data/MNIST/t10k-images-idx3-ubyte.gz
Extracting data/MNIST/t10k-labels-idx1-ubyte.gz
```

现在已经载入了MNIST数据集，它由70,000张图像和对应的标签（比如图像的类别）组成。数据集分成三份互相独立的子集。我们在教程中只用训练集和测试集。

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(data.train.labels)))
print("- Test-set:\t\t{}".format(len(data.test.labels)))
print("- Validation-set:\t{}".format(len(data.validation.labels)
))
```

```
Size of:
- Training-set:		55000
- Test-set:		10000
- Validation-set:	5000
```

类型标签使用One-Hot编码，这意外每个标签是长为10的向量，除了一个元素之外，其他的都为零。这个元素的索引就是类别的数字，即相应图片中画的数字。我们也需要测试数据集类别数字的整型值，用下面的方法来计算。

```
data.test.cls = np.argmax(data.test.labels, axis=1)
```

数据维度

在下面的源码中，有很多地方用到了数据维度。它们只在一个地方定义，因此我们可以在代码中使用这些数字而不是直接写数字。

```
# We know that MNIST images are 28 pixels in each dimension.
img_size = 28

# Images are stored in one-dimensional arrays of this length.
img_size_flat = img_size * img_size

# Tuple with height and width of images used to reshape arrays.
img_shape = (img_size, img_size)

# Number of colour channels for the images: 1 channel for gray-s
cale.
num_channels = 1

# Number of classes, one class for each of 10 digits.
num_classes = 10
```

用来绘制图片的帮助函数

这个函数用来在3x3的栅格中画9张图像，然后在每张图像下面写出真实类别和预测类别。

```
def plot_images(images, cls_true, cls_pred=None):
    assert len(images) == len(cls_true) == 9

    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_shape), cmap='binary')

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i],
cls_pred[i])

        # Show the classes as the label on the x-axis.
        ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

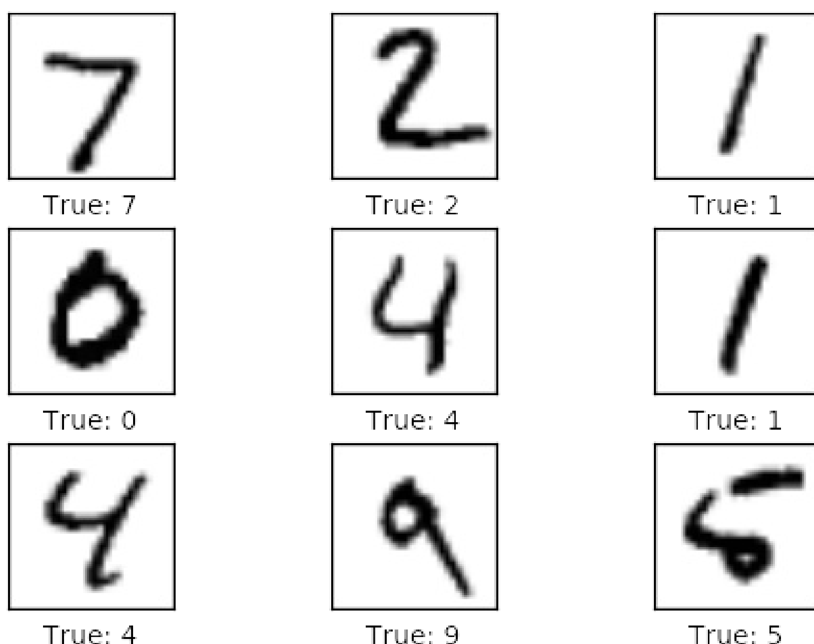
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

绘制几张图像来看看数据是否正确

```
# Get the first images from the test-set.
images = data.test.images[0:9]

# Get the true classes for those images.
cls_true = data.test.cls[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true)
```



TensorFlow图

TensorFlow的全部目的就是使用一个称之为计算图（computational graph）的东西，它会比直接在Python中进行相同计算量要高效得多。TensorFlow比Numpy更高效，因为TensorFlow了解整个需要运行的计算图，然而Numpy只知道某个时间点上唯一的数学运算。

TensorFlow也能够自动地计算需要优化的变量的梯度，使得模型有更好的表现。这是由于图是简单数学表达式的结合，因此整个图的梯度可以用链式法则推导出来。

TensorFlow还能利用多核CPU和GPU，Google也为TensorFlow制造了称为TPUs（Tensor Processing Units）的特殊芯片，它比GPU更快。

一个TensorFlow图由下面几个部分组成，后面会详细描述：

- 占位符变量（Placeholder）用来改变图的输入。
- 模型变量（Model）将会被优化，使得模型表现得更好。
- 模型本质上就是一些数学函数，它根据Placeholder和模型的输入变量来计算一些输出。
- 一个cost度量用来指导变量的优化。
- 一个优化策略会更新模型的变量。

另外，TensorFlow图也包含了一些调试状态，比如用TensorBoard打印log数据，本教程不涉及这些。

占位符 （Placeholder）变量

Placeholder是作为图的输入，我们每次运行图的时候都可能改变它们。将这个过程称为feeding placeholder变量，后面将会描述这个。

首先我们为输入图像定义placeholder变量。这让我们可以改变输入到TensorFlow图中的图像。这也是一个张量（tensor），代表一个多维向量或矩阵。数据类型设置为 `float32` ，形状设为 `[None, img_size_flat]` ， `None` 代表tensor可能保存着任意数量的图像，每张图象是一个长度为 `img_size_flat` 的向量。

```
x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name=
'x')
```

卷积层希望x被编码为4维张量，因此我们需要将它的形状转换至 `[num_images, img_height, img_width, num_channels]` 。注意 `img_height == img_width == img_size` ，如果第一维的大小设为-1， `num_images` 的大小也会被自动推导出来。转换运算如下：

```
x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
```

接下来我们为输入变量 `x` 中的图像所对应的真实标签定义placeholder变量。变量的形状是 `[None, num_classes]` ，这代表着它保存了任意数量的标签，每个标签是长度为 `num_classes` 的向量，本例中长度为10。

```
y_true = tf.placeholder(tf.float32, shape=[None, 10], name='y_tr
ue')
```

我们也可以为class-number提供一个placeholder，但这里用argmax来计算它。这里只是TensorFlow中的一些操作，没有执行什么运算。

```
y_true_cls = tf.argmax(y_true, dimension=1)
```

TensorFlow 实现

这一节显示了教程 #02 中直接用TensorFlow实现卷积神经网络的源代码。这份Notebook中并没有直接用到这些代码，只是为了方便和下面PrettyTensor的实现进行比较。

这里要注意的是有多少代码量以及TensorFlow保存数据、进行运算的底层细节。即使在很小的神经网络中也容易犯错。

帮助函数

在直接用TensorFlow实现时，我们创建一些在构造图时常用到的帮助函数。

这两个函数在TensorFlow图中创建新的变量并用随机值初始化。

```
def new_weights(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
```

```
def new_biases(length):
    return tf.Variable(tf.constant(0.05, shape=[length]))
```

下面的帮助函数创建一个新的卷积网络。输入和输出是4维的张量（4-rank tensors）。注意TensorFlow API的底层细节，比如权重变量的大小。这里很容易犯错，可能会导致奇怪的错误信息，并且很难调试。

```
def new_conv_layer(input,              # The previous layer.
                   num_input_channels, # Num. channels in prev.
layer.
                   filter_size,        # Width and height of fil
ters.
                   num_filters,        # Number of filters.
                   use_pooling=True):  # Use 2x2 max-pooling.

    # Shape of the filter-weights for the convolution.
    # This format is determined by the TensorFlow API.
    shape = [filter_size, filter_size, num_input_channels, num_f
ilters]

    # Create new weights aka. filters with the given shape.
    weights = new_weights(shape=shape)

    # Create new biases, one for each filter.
    biases = new_biases(length=num_filters)

    # Create the TensorFlow operation for convolution.
    # Note the strides are set to 1 in all dimensions.
    # The first and last stride must always be 1,
    # because the first is for the image-number and
    # the last is for the input-channel.
    # But e.g. strides=[1, 2, 2, 1] would mean that the filter
    # is moved 2 pixels across the x- and y-axis of the image.
    # The padding is set to 'SAME' which means the input image
    # is padded with zeroes so the size of the output is the sam
e.
    layer = tf.nn.conv2d(input=input,
                         filter=weights,
                         strides=[1, 1, 1, 1],
                         padding='SAME')

    # Add the biases to the results of the convolution.
    # A bias-value is added to each filter-channel.
    layer += biases

    # Use pooling to down-sample the image resolution?
```

```
    if use_pooling:
        # This is 2x2 max-pooling, which means that we
        # consider 2x2 windows and select the largest value
        # in each window. Then we move 2 pixels to the next wind
ow.
        layer = tf.nn.max_pool(value=layer,
                               ksize=[1, 2, 2, 1],
                               strides=[1, 2, 2, 1],
                               padding='SAME')

    # Rectified Linear Unit (ReLU).
    # It calculates max(x, 0) for each input pixel x.
    # This adds some non-linearity to the formula and allows us
    # to learn more complicated functions.
    layer = tf.nn.relu(layer)

    # Note that ReLU is normally executed before the pooling,
    # but since relu(max_pool(x)) == max_pool(relu(x)) we can
    # save 75% of the relu-operations by max-pooling first.

    # We return both the resulting layer and the filter-weights
    # because we will plot the weights later.
    return layer, weights
```

下面的帮助函数将一个4维张量转换到2维，因此我们可以在卷积层之后添加一个全连接层。

```
def flatten_layer(layer):
    # Get the shape of the input layer.
    layer_shape = layer.get_shape()

    # The shape of the input layer is assumed to be:
    # layer_shape == [num_images, img_height, img_width, num_cha
nnels]

    # The number of features is: img_height * img_width * num_ch
annels
    # We can use a function from TensorFlow to calculate this.
    num_features = layer_shape[1:4].num_elements()

    # Reshape the layer to [num_images, num_features].
    # Note that we just set the size of the second dimension
    # to num_features and the size of the first dimension to -1
    # which means the size in that dimension is calculated
    # so the total size of the tensor is unchanged from the resh
aping.
    layer_flat = tf.reshape(layer, [-1, num_features])

    # The shape of the flattened layer is now:
    # [num_images, img_height * img_width * num_channels]

    # Return both the flattened layer and the number of features.

    return layer_flat, num_features
```

接下来的帮助函数创建一个全连接层。

```
def new_fc_layer(input,          # The previous layer.
                 num_inputs,     # Num. inputs from prev. layer.
                 num_outputs,    # Num. outputs.
                 use_relu=True): # Use Rectified Linear Unit (Re
LU)?

    # Create new weights and biases.
    weights = new_weights(shape=[num_inputs, num_outputs])
    biases = new_biases(length=num_outputs)

    # Calculate the layer as the matrix multiplication of
    # the input and weights, and then add the bias-values.
    layer = tf.matmul(input, weights) + biases

    # Use ReLU?
    if use_relu:
        layer = tf.nn.relu(layer)

    return layer
```

构造图（Graph）

现在将会用上面的帮助函数来创建卷积神经网络。如果没有这些函数的话，代码将会又长又难以理解。

注意，我们并不会运行下面的代码。写在这里只是为了与PrettyTensor的代码进行比较。

之前的教程使用定义好的常量，因此很容易改变（变量）。比如，我们没有将 `filter_size=5` 当作 `new_conv_layer()` 的参数，而是令 `filter_size=filter_size1` ，然后在其他地方定义 `filter_size1=5` 。这样子就很容易改变所有的常量。

```
if False:  # Don't execute this! Just show it for easy compariso
n.
    # First convolutional layer.
    layer_conv1, weights_conv1 = \
        new_conv_layer(input=x_image,
                       num_input_channels=num_channels,
                       filter_size=5,
                       num_filters=16,
                       use_pooling=True)

    # Second convolutional layer.
    layer_conv2, weights_conv2 = \
        new_conv_layer(input=layer_conv1,
                       num_input_channels=16,
                       filter_size=5,
                       num_filters=36,
                       use_pooling=True)

    # Flatten layer.
    layer_flat, num_features = flatten_layer(layer_conv2)

    # First fully-connected layer.
    layer_fc1 = new_fc_layer(input=layer_flat,
                             num_inputs=num_features,
                             num_outputs=128,
                             use_relu=True)

    # Second fully-connected layer.
    layer_fc2 = new_fc_layer(input=layer_fc1,
                             num_inputs=128,
                             num_outputs=num_classes,
                             use_relu=False)

    # Predicted class-label.
    y_pred = tf.nn.softmax(layer_fc2)

    # Cross-entropy for the classification of each image.
    cross_entropy = \
        tf.nn.softmax_cross_entropy_with_logits(logits=layer_fc2
,
                                                labels=y_true)

    # Loss aka. cost-measure.
    # This is the scalar value that must be minimized.
    loss = tf.reduce_mean(cross_entropy)
```

PrettyTensor实现

这一节演示如何用PrettyTensor来实现一个相同的卷积神经网络。

基本思想就是用一个PrettyTensor object封装输入张量 `x_image` ，它有一个添加新卷积层的帮助函数，以此来创建整个神经网络。这有点像我们之前实现的那些帮助函数，但它更简单一些，因为PrettyTensor记录每一层的输入和输出维度等等。

```
x_pretty = pt.wrap(x_image)
```

现在我们已经将输入图像装到一个PrettyTensor的object中，再用几行代码就可以添加卷积层和全连接层。

注意，在 `with` 代码块中， `pt.defaults_scope(activation_fn=tf.nn.relu)` 把 `activation_fn=tf.nn.relu` 当作每个的层参数，因此这些层都用到了 Rectified Linear Units (ReLU) 。 `defaults_scope` 使我们能更方便地修改所有层的参数。

```
with pt.defaults_scope(activation_fn=tf.nn.relu):
    y_pred, loss = x_pretty.\
        conv2d(kernel=5, depth=16, name='layer_conv1').\
        max_pool(kernel=2, stride=2).\
        conv2d(kernel=5, depth=36, name='layer_conv2').\
        max_pool(kernel=2, stride=2).\
        flatten().\
        fully_connected(size=128, name='layer_fc1').\
        softmax_classifier(num_classes=num_classes, labels=y_tru
e)
```

就是这样！现在我们用几行代码就创建了一个完全一样的卷积神经网络，如果直接用TensorFlow实现的话需要一大段非常复杂的代码。

用PrettyTensor来代替TensorFlow，我们可以清楚地看到网络的构造以及数据如何在网络中流通。这让我们可以专注于神经网络的关键思想而不是底层的实现细节。它十分简单优雅！

获取权重

不幸的是，使用PrettyTensor时并非所有的事都那么优雅。

下面，我们想要绘制出卷积层的权重。在用TensorFlow实现时，我们自己创建了变量，所以可以直接访问它们。但使用PrettyTensor构造网络时，所有的变量都是间接地由PrettyTensor创建。因此我们不得不从TensorFlow中找回变量。

我们用 `layer_conv1` 和 `layer_conv2` 代表两个卷积层。这也叫变量作用域（不要与上面描述的 `defaults_scope` 混淆了）。PrettyTensor会自动给它为每个层创建的变量命名，因此我们可以通过层的作用域名称和变量名来取得某一层的权重。

函数实现有点笨拙，因为我们不得不用TensorFlow函数 `get_variable()` ，它是设计给其他用途的，创建新的变量或重用现有变量。创建下面的帮助函数很简单。

```
def get_weights_variable(layer_name):
    # Retrieve an existing variable named 'weights' in the scope
    # with the given layer_name.
    # This is awkward because the TensorFlow function was
    # really intended for another purpose.

    with tf.variable_scope(layer_name, reuse=True):
        variable = tf.get_variable('weights')

    return variable
```

借助这个帮助函数我们可以获取变量。这些是TensorFlow的objects。你需要类似的操作来获取变量的内容： `contents = session.run(weights_conv1)` ，下面会提到这个。

```
weights_conv1 = get_weights_variable(layer_name='layer_conv1')
weights_conv2 = get_weights_variable(layer_name='layer_conv2')
```

优化方法

PrettyTensor给我们提供了预测类型标签(`y_pred`)以及一个需要最小化的损失度量，用来提升神经网络分类图片的能力。

PrettyTensor的文档并没有说明它的损失度量是用cross-entropy还是其他的。但现在我们用 `AdamOptimizer` 来最小化损失。

优化过程并不是在这里执行。实际上，还没计算任何东西，我们只是往TensorFlow图中添加了优化器，以便后续操作。

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(
loss)
```

性能度量

我们需要另外一些性能度量，来向用户展示这个过程。

首先我们从神经网络输出的 `y_pred` 中计算出预测的类别，它是一个包含10个元素的向量。类别数字是最大元素的索引。

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

然后创建一个布尔向量，用来告诉我们每张图片的真实类别是否与预测类别相同。

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

上面的计算先将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确度。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
))
```

运行TensorFlow

创建TensorFlow会话（session）

一旦创建了TensorFlow图，我们需要创建一个TensorFlow会话，用来运行图。

```
session = tf.Session()
```

初始化变量

我们需要在开始优化 `weights` 和 `biases` 变量之前对它们进行初始化。

```
session.run(tf.global_variables_initializer())
```

用来优化迭代的帮助函数

在训练集中有50,000张图。用这些图像计算模型的梯度会花很多时间。因此我们利用随机梯度下降的方法，它在优化器的每次迭代里只用到了一小部分的图像。 如果内存耗尽导致电脑死机或变得很慢，你应该试着减少这些数量，但同时可能还需要更优化的迭代。

```
train_batch_size = 64
```

函数执行了多次的优化迭代来逐步地提升网络层的变量。在每次迭代中，从训练集中选择一批新的数据，然后TensorFlow用这些训练样本来执行优化器。每100次迭代会打印出相关信息。

```
# Counter for total number of iterations performed so far.
total_iterations = 0

def optimize(num_iterations):
    # Ensure we update the global variable rather than a local c
```

```
opy.
    global total_iterations

    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(total_iterations,
                   total_iterations + num_iterations):

        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = data.train.next_batch(train_batc
h_size)

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimiz
er.
        session.run(optimizer, feed_dict=feed_dict_train)

        # Print status every 100 iterations.
        if i % 100 == 0:
            # Calculate the accuracy on the training-set.
            acc = session.run(accuracy, feed_dict=feed_dict_trai
n)

            # Message for printing.
            msg = "Optimization Iteration: {0:>6}, Training Accu
racy: {1:>6.1%}"

            # Print it.
            print(msg.format(i + 1, acc))

    # Update the total number of iterations performed.
    total_iterations += num_iterations

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_
dif)))))
```

用来绘制错误样本的帮助函数

函数用来绘制测试集中被误分类的样本。

```
def plot_example_errors(cls_pred, correct):
    # This function is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # correct is a boolean array whether the predicted class
    # is equal to the true class for each image in the test-set.

    # Negate the boolean array.
    incorrect = (correct == False)

    # Get the images from the test-set that have been
    # incorrectly classified.
    images = data.test.images[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = data.test.cls[incorrect]

    # Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])
```

绘制混淆（confusion）矩阵的帮助函数

```
def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the true classifications for the test-set.
    cls_true = data.test.cls

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_true,
                          y_pred=cls_pred)

    # Print the confusion matrix as text.
    print(cm)

    # Plot the confusion matrix as an image.
    plt.matshow(cm)

    # Make various adjustments to the plot.
    plt.colorbar()
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
    plt.yticks(tick_marks, range(num_classes))
    plt.xlabel('Predicted')
    plt.ylabel('True')

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

展示性能的帮助函数

函数用来打印测试集上的分类准确度。

为测试集上的所有图片计算分类会花费一段时间，因此我们直接用这个函数来调用上面的结果，这样就不用每次都重新计算了。

这个函数可能会占用很多电脑内存，这也是为什么将测试集分成更小的几个部分。如果你的电脑内存比较小或死机了，就要试着降低batch-size。

```
# Split the test-set into smaller batches of this size.
test_batch_size = 256

def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # Number of images in the test-set.
    num_test = len(data.test.images)
```

```
    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_test, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing thi
s.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_test:
        # The ending index for the next batch is denoted j.
        j = min(i + test_batch_size, num_test)

        # Get the images from the test-set between index i and j.

        images = data.test.images[i:j, :]

        # Get the associated labels.
        labels = data.test.labels[i:j, :]

        # Create a feed-dict with these images and labels.
        feed_dict = {x: images,
                     y_true: labels}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_d
ict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Convenience variable for the true class-numbers of the tes
t-set.
    cls_true = data.test.cls

    # Create a boolean array whether each image is correctly cla
ssified.
    correct = (cls_true == cls_pred)

    # Calculate the number of correctly classified images.
    # When summing a boolean array, False means 0 and True means
 1.
    correct_sum = correct.sum()

    # Classification accuracy is the number of correctly classif
ied
    # images divided by the total number of images in the test-s
et.
    acc = float(correct_sum) / num_test
```

```
    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, correct_sum, num_test))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

优化之前的性能

测试集上的准确度很低，这是由于模型只做了初始化，并没做任何优化，所以它只是对图像做随机分类。

```
print_test_accuracy()
```

```
Accuracy on Test-Set: 9.1% (909 / 10000)
```

1次迭代后的性能

做了一次优化后，此时优化器的学习率很低，性能其实并没有多大提升。

```
optimize(num_iterations=1)
```

```
Optimization Iteration:      1, Training Accuracy:   6.2%
Time usage: 0:00:00
```

```
print_test_accuracy()
```

```
Accuracy on Test-Set: 8.9% (892 / 10000)
```

100次迭代优化后的性能

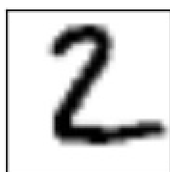
100次优化迭代之后，模型显著地提升了分类的准确度。

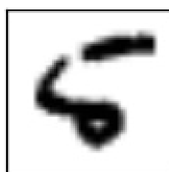
```
optimize(num_iterations=99) # We already performed 1 iteration a
bove.
```

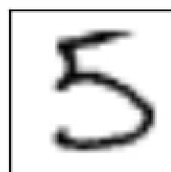
```
Time usage: 0:00:00
```

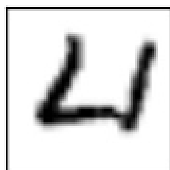
```
print_test_accuracy(show_example_errors=True)
```

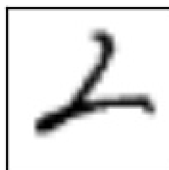
```
Accuracy on Test-Set: 83.9% (8393 / 10000)
Example errors:
```


True: 2, Pred: 3

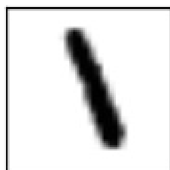

True: 5, Pred: 4


True: 5, Pred: 3


True: 4, Pred: 0


True: 2, Pred: 6


True: 5, Pred: 3


True: 1, Pred: 3

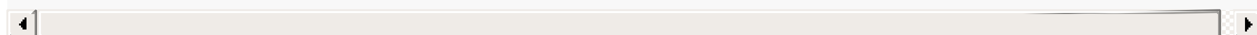

True: 5, Pred: 3


True: 6, Pred: 2

1000次优化迭代后的性能

1000次优化迭代之后，模型在测试集上的准确度超过了90%。

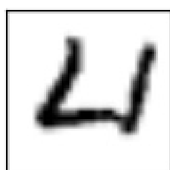
```
optimize(num_iterations=900) # We performed 100 iterations above.
```

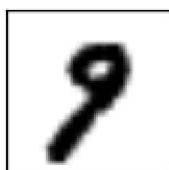


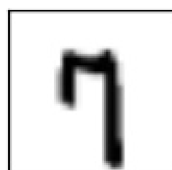
```
Optimization Iteration:    101, Training Accuracy:  93.8%
Optimization Iteration:    201, Training Accuracy:  89.1%
Optimization Iteration:    301, Training Accuracy:  85.9%
Optimization Iteration:    401, Training Accuracy:  87.5%
Optimization Iteration:    501, Training Accuracy:  92.2%
Optimization Iteration:    601, Training Accuracy:  95.3%
Optimization Iteration:    701, Training Accuracy:  95.3%
Optimization Iteration:    801, Training Accuracy:  90.6%
Optimization Iteration:    901, Training Accuracy:  98.4%
Time usage: 0:00:03
```

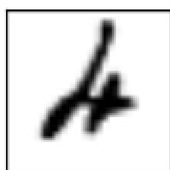
```
print_test_accuracy(show_example_errors=True)
```

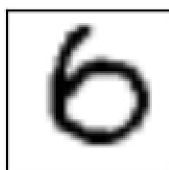
```
Accuracy on Test-Set: 96.3% (9634 / 10000)
Example errors:
```

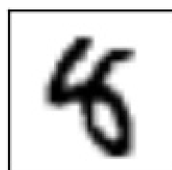

True: 4, Pred: 6

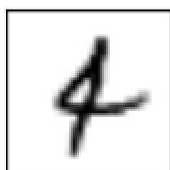

True: 9, Pred: 7

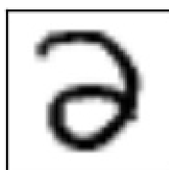

True: 7, Pred: 9

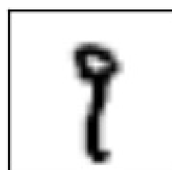

True: 4, Pred: 6


True: 6, Pred: 0


True: 8, Pred: 4


True: 4, Pred: 6


True: 2, Pred: 3


True: 9, Pred: 1

10,000次优化迭代后的性能

经过10,000次优化迭代后，测试集上的分类准确率高达99%。

```
optimize(num_iterations=9000) # We performed 1000 iterations abo
ve.
```

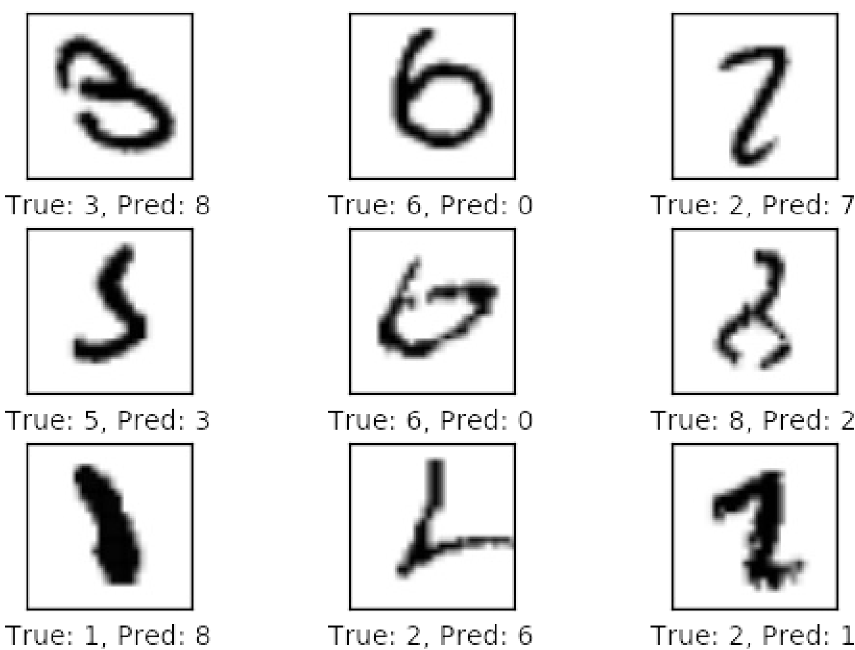
```
Optimization Iteration:   1001, Training Accuracy:  98.4%
```

```
Optimization Iteration:   1101, Training Accuracy:  95.3%
Optimization Iteration:   1201, Training Accuracy:  98.4%
Optimization Iteration:   1301, Training Accuracy:  96.9%
Optimization Iteration:   1401, Training Accuracy: 100.0%
Optimization Iteration:   1501, Training Accuracy:  95.3%
Optimization Iteration:   1601, Training Accuracy:  96.9%
Optimization Iteration:   1701, Training Accuracy:  96.9%
Optimization Iteration:   1801, Training Accuracy:  98.4%
Optimization Iteration:   1901, Training Accuracy:  96.9%
Optimization Iteration:   2001, Training Accuracy:  98.4%
Optimization Iteration:   2101, Training Accuracy:  95.3%
Optimization Iteration:   2201, Training Accuracy:  98.4%
Optimization Iteration:   2301, Training Accuracy:  98.4%
Optimization Iteration:   2401, Training Accuracy:  98.4%
Optimization Iteration:   2501, Training Accuracy:  93.8%
Optimization Iteration:   2601, Training Accuracy:  98.4%
Optimization Iteration:   2701, Training Accuracy:  98.4%
Optimization Iteration:   2801, Training Accuracy:  95.3%
Optimization Iteration:   2901, Training Accuracy:  98.4%
Optimization Iteration:   3001, Training Accuracy:  98.4%
Optimization Iteration:   3101, Training Accuracy: 100.0%
Optimization Iteration:   3201, Training Accuracy:  96.9%
Optimization Iteration:   3301, Training Accuracy: 100.0%
Optimization Iteration:   3401, Training Accuracy:  98.4%
Optimization Iteration:   3501, Training Accuracy:  96.9%
Optimization Iteration:   3601, Training Accuracy:  98.4%
Optimization Iteration:   3701, Training Accuracy:  96.9%
Optimization Iteration:   3801, Training Accuracy: 100.0%
Optimization Iteration:   3901, Training Accuracy:  98.4%
Optimization Iteration:   4001, Training Accuracy:  96.9%
Optimization Iteration:   4101, Training Accuracy:  98.4%
Optimization Iteration:   4201, Training Accuracy: 100.0%
Optimization Iteration:   4301, Training Accuracy: 100.0%
Optimization Iteration:   4401, Training Accuracy: 100.0%
Optimization Iteration:   4501, Training Accuracy: 100.0%
Optimization Iteration:   4601, Training Accuracy:  98.4%
Optimization Iteration:   4701, Training Accuracy:  96.9%
Optimization Iteration:   4801, Training Accuracy:  95.3%
Optimization Iteration:   4901, Training Accuracy: 100.0%
Optimization Iteration:   5001, Training Accuracy:  96.9%
Optimization Iteration:   5101, Training Accuracy: 100.0%
Optimization Iteration:   5201, Training Accuracy:  98.4%
Optimization Iteration:   5301, Training Accuracy:  98.4%
Optimization Iteration:   5401, Training Accuracy: 100.0%
Optimization Iteration:   5501, Training Accuracy:  98.4%
Optimization Iteration:   5601, Training Accuracy:  96.9%
Optimization Iteration:   5701, Training Accuracy: 100.0%
Optimization Iteration:   5801, Training Accuracy:  96.9%
Optimization Iteration:   5901, Training Accuracy: 100.0%
Optimization Iteration:   6001, Training Accuracy:  98.4%
Optimization Iteration:   6101, Training Accuracy:  98.4%
Optimization Iteration:   6201, Training Accuracy:  98.4%
Optimization Iteration:   6301, Training Accuracy:  98.4%
```

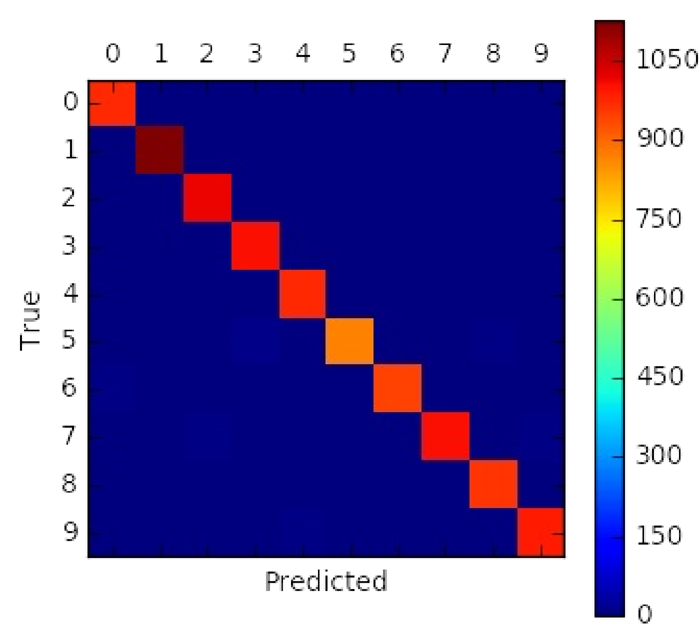
```
Optimization Iteration:   6401, Training Accuracy: 100.0%
Optimization Iteration:   6501, Training Accuracy: 100.0%
Optimization Iteration:   6601, Training Accuracy: 100.0%
Optimization Iteration:   6701, Training Accuracy: 100.0%
Optimization Iteration:   6801, Training Accuracy:  96.9%
Optimization Iteration:   6901, Training Accuracy: 100.0%
Optimization Iteration:   7001, Training Accuracy: 100.0%
Optimization Iteration:   7101, Training Accuracy: 100.0%
Optimization Iteration:   7201, Training Accuracy: 100.0%
Optimization Iteration:   7301, Training Accuracy:  96.9%
Optimization Iteration:   7401, Training Accuracy: 100.0%
Optimization Iteration:   7501, Training Accuracy: 100.0%
Optimization Iteration:   7601, Training Accuracy:  96.9%
Optimization Iteration:   7701, Training Accuracy: 100.0%
Optimization Iteration:   7801, Training Accuracy: 100.0%
Optimization Iteration:   7901, Training Accuracy: 100.0%
Optimization Iteration:   8001, Training Accuracy:  98.4%
Optimization Iteration:   8101, Training Accuracy: 100.0%
Optimization Iteration:   8201, Training Accuracy: 100.0%
Optimization Iteration:   8301, Training Accuracy: 100.0%
Optimization Iteration:   8401, Training Accuracy: 100.0%
Optimization Iteration:   8501, Training Accuracy:  98.4%
Optimization Iteration:   8601, Training Accuracy: 100.0%
Optimization Iteration:   8701, Training Accuracy: 100.0%
Optimization Iteration:   8801, Training Accuracy: 100.0%
Optimization Iteration:   8901, Training Accuracy: 100.0%
Optimization Iteration:   9001, Training Accuracy:  98.4%
Optimization Iteration:   9101, Training Accuracy:  98.4%
Optimization Iteration:   9201, Training Accuracy: 100.0%
Optimization Iteration:   9301, Training Accuracy: 100.0%
Optimization Iteration:   9401, Training Accuracy:  98.4%
Optimization Iteration:   9501, Training Accuracy: 100.0%
Optimization Iteration:   9601, Training Accuracy: 100.0%
Optimization Iteration:   9701, Training Accuracy: 100.0%
Optimization Iteration:   9801, Training Accuracy:  98.4%
Optimization Iteration:   9901, Training Accuracy: 100.0%
Time usage: 0:00:27
```

```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

```
Accuracy on Test-Set: 98.8% (9881 / 10000)
Example errors:
```




```
Confusion Matrix:
[[ 975    0    0    0    0    0    1    1    3    0]
 [   0 1127    2    0    0    0    1    2    3    0]
 [   2    2 1019    1    1    0    1    2    4    0]
 [   0    0    0 1005    0    1    0    1    3    0]
 [   0    0    0    0  977    0    1    0    1    3]
 [   2    0    0   13    0  870    1    0    6    0]
 [   5    2    0    0    1    3  943    0    4    0]
 [   0    2    8    2    1    0    0 1007    1    7]
 [   2    0    2    3    1    1    0    0  964    1]
 [   0    2    0    4    5    1    0    1    2  994]]
```



权重和层的可视化

当我们直接用TensorFlow来实现卷积神经网络时，可以很容易地画出卷积权重和不同层的输出图像。当使用PrettyTensor的时候，我们也可以通过上面提到过的方法取得权重，但我们无法简单得到卷积层的输出（图像）。因此下面只绘制了权重。

绘制卷积权重的帮助函数

```
def plot_conv_weights(weights, input_channel=0):
    # Assume weights are TensorFlow ops for 4-dim variables
    # e.g. weights_conv1 or weights_conv2.

    # Retrieve the values of the weight-variables from TensorFlo
w.
    # A feed-dict is not necessary because nothing is calculated.

    w = session.run(weights)

    # Get the lowest and highest values for the weights.
    # This is used to correct the colour intensity across
    # the images so they can be compared with each other.
    w_min = np.min(w)
    w_max = np.max(w)

    # Number of filters used in the conv. layer.
    num_filters = w.shape[3]

    # Number of grids to plot.
    # Rounded-up, square-root of the number of filters.
    num_grids = math.ceil(math.sqrt(num_filters))

    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(num_grids, num_grids)

    # Plot all the filter-weights.
    for i, ax in enumerate(axes.flat):
        # Only plot the valid filter-weights.
        if i<num_filters:
            # Get the weights for the i'th filter of the input c
hannel.
            # See new_conv_layer() for details on the format
            # of this 4-dim tensor.
            img = w[:, :, input_channel, i]

            # Plot image.
            ax.imshow(img, vmin=w_min, vmax=w_max,
                      interpolation='nearest', cmap='seismic')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

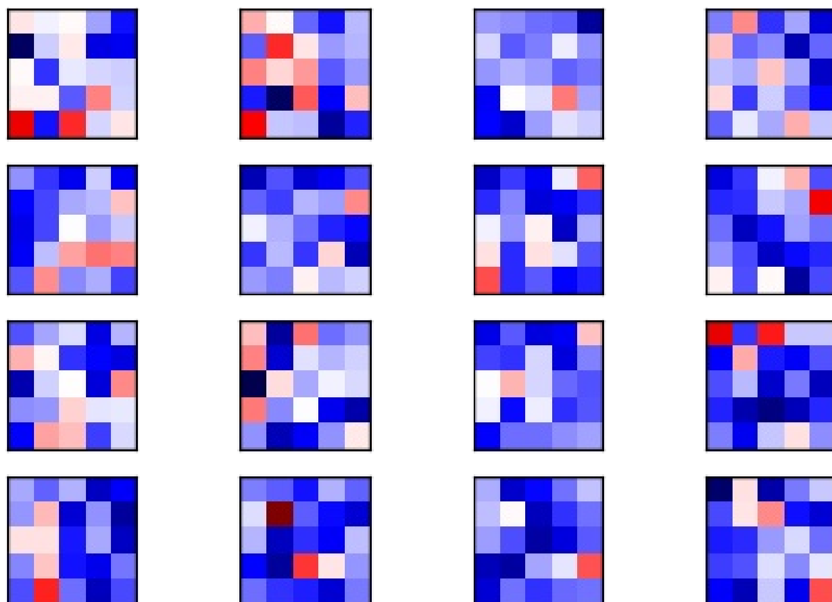
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

卷积层 1

现在绘制第一个卷积层的滤波权重。

其中正值权重是红色的，负值为蓝色。

```
plot_conv_weights(weights=weights_conv1)
```



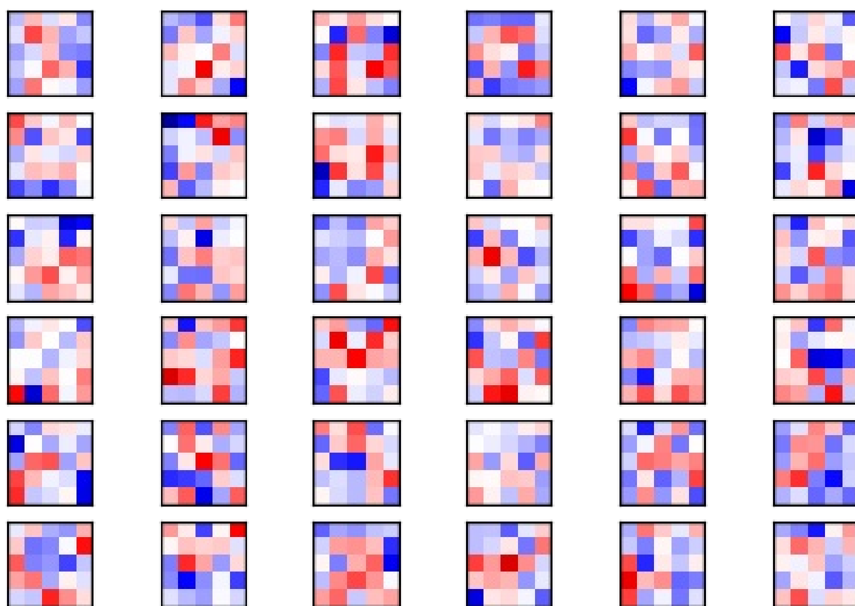
卷积层 2

现在绘制第二个卷积层的滤波权重。

第一个卷积层有16个输出通道，代表着第二个卷基层有16个输入。第二个卷积层的每个输入通道也有一些权重滤波。我们先绘制第一个通道的权重滤波。

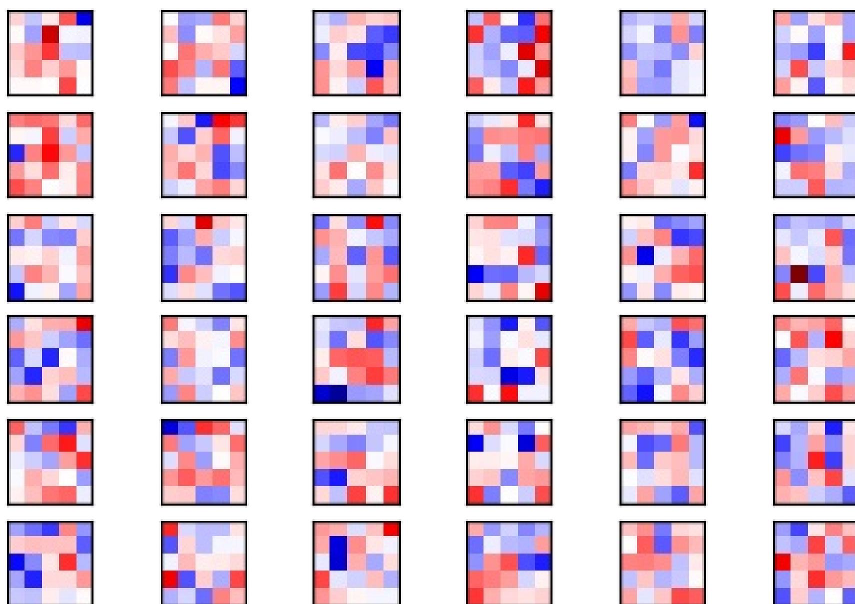
同样的，正值是红色，负值是蓝色。

```
plot_conv_weights(weights=weights_conv2, input_channel=0)
```



第二个卷积层共有16个输入通道，我们可以同样地画出15张其他滤波权重图像。这里我们再画一下第二个通道的图像。

```
plot_conv_weights(weights=weights_conv2, input_channel=1)
```



关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# session.close()
```

总结

相比直接使用TensorFlow，PrettyTensor可以用更简单的代码来实现神经网络。这使你能够专注于自己的想法而不是底层的实现细节。它让代码更易于理解，也减少犯错的可能。

然而，PrettyTensor中有一些矛盾和笨拙的设计，它的文档简短而又令人疑惑，也不易于学习。希望未来会有所改进（本文写于2016七月）。

还有一些PrettyTensor的替代品，包括TFLearn和Keras。

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 将所有层的激活函数改成sigmod。
- 在一些层中使用sigmod，一些层中使用relu。这里能用 `defaults_scope` 吗？
- 在所有层里使用12loss。然后试着只在某些层里使用这个。
- 用PrettyTensor的reshape函数代替TensorFlow的。其中某一个会更好吗？
- 在全连接层后面添加一个dropout-layer。如果你在训练和测试的时候想要一个不同的 `keep_prob` ，就需要在feed-dict中设置一个placeholder变量。
- 用stride=2来代替 2x2 max-pooling层。分类准确率会有所不同么？你多次优化它们之后呢？差异是随机的，你如何度量是否真实存在差异呢？在卷积层中使用max-pooling和stride的优缺点是什么？
- 改变层的参数，比如kernel、depth、size等等。耗费的时间以及分类准确率有什么差别？
- 添加或删除某些卷积层和全连接层。
- 你设计的表现良好的最简网络是什么？
- 取回卷积层的bias-values并打印出来。参考一下 `get_weights_variable()` 的实现。
- 不看源码，自己重写程序。
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #04

保存 & 恢复

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

简介

这篇教程展示了如何保存以及恢复神经网络中的变量。在优化的过程中，当验证集上分类准确率提高时，保存神经网络的变量。如果经过1000次迭代还不能提升性能时，就终止优化。然后我们重新载入在验证集上表现最好的变量。

这种策略称为Early-Stopping。它用来避免神经网络的过拟合。（过拟合）会在神经网络训练时间太长时出现，此时神经网络开始学习训练集中的噪声，将导致它误分类新的图像。

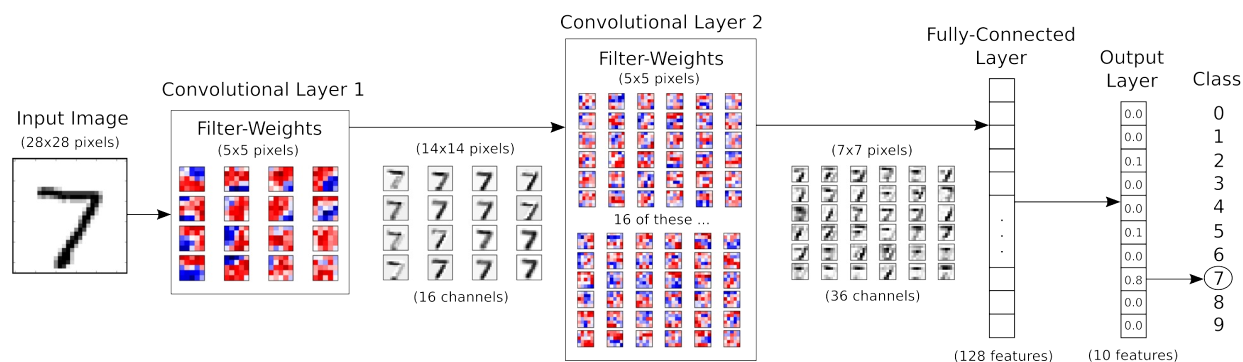
这篇教程主要是用神经网络来识别MNIST数据集中的手写数字，过拟合在这里并不是什么大问题。但本教程展示了Early Stopping的思想。

本文基于上一篇教程，你需要了解基本的TensorFlow和附加包Pretty Tensor。其中大量代码和文字与之前教程相似，如果你已经看过就可以快速地浏览本文。

流程图

下面的图表直接显示了之后实现的卷积神经网络中数据的传递。网络有两个卷积层和两个全连接层，最后一层是用来给输入图像分类的。关于网络和卷积的更多细节描述见教程 #02 。

```
from IPython.display import Image
Image('images/02_network_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
import time
from datetime import timedelta
import math
import os

# Use PrettyTensor to simplify Neural Network construction.
import prettytensor as pt
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.12.0-rc0'
```

PrettyTensor 版本:

```
pt.__version__
```

```
'0.7.1'
```

载入数据

MNIST数据集大约12MB，如果没在给定路径中找到就会自动下载。

```
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets('data/MNIST/', one_hot=True)
```

```
Extracting data/MNIST/train-images-idx3-ubyte.gz
Extracting data/MNIST/train-labels-idx1-ubyte.gz
Extracting data/MNIST/t10k-images-idx3-ubyte.gz
Extracting data/MNIST/t10k-labels-idx1-ubyte.gz
```

现在已经载入了MNIST数据集，它由70,000张图像和对应的标签（比如图像的类别）组成。数据集分成三份互相独立的子集。我们在教程中只用训练集和测试集。

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(data.train.labels)))
print("- Test-set:\t\t{}".format(len(data.test.labels)))
print("- Validation-set:\t{}".format(len(data.validation.labels)
))
```

```
Size of:
- Training-set:		55000
- Test-set:		10000
- Validation-set:	5000
```

类型标签使用One-Hot编码，这意外每个标签是长为10的向量，除了一个元素之外，其他的都为零。这个元素的索引就是类别的数字，即相应图片中画的数字。我们也需要测试数据集类别数字的整型值，用下面的方法来计算。

```
data.test.cls = np.argmax(data.test.labels, axis=1)
data.validation.cls = np.argmax(data.validation.labels, axis=1)
```

数据维度

在下面的源码中，有很多地方用到了数据维度。它们只在一个地方定义，因此我们可以在代码中使用这些数字而不是直接写数字。

```
# We know that MNIST images are 28 pixels in each dimension.
img_size = 28

# Images are stored in one-dimensional arrays of this length.
img_size_flat = img_size * img_size

# Tuple with height and width of images used to reshape arrays.
img_shape = (img_size, img_size)

# Number of colour channels for the images: 1 channel for gray-s
cale.
num_channels = 1

# Number of classes, one class for each of 10 digits.
num_classes = 10
```

用来绘制图片的帮助函数

这个函数用来在3x3的栅格中画9张图像，然后在每张图像下面写出真实类别和预测类别。

```
def plot_images(images, cls_true, cls_pred=None):
    assert len(images) == len(cls_true) == 9

    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_shape), cmap='binary')

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i],
cls_pred[i])

        # Show the classes as the label on the x-axis.
        ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

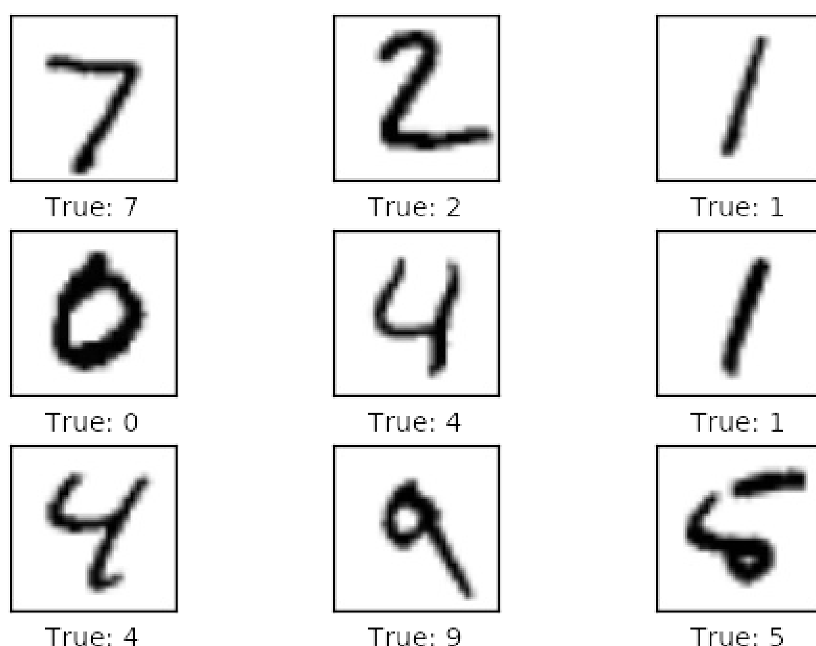
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

绘制几张图像来看看数据是否正确

```
# Get the first images from the test-set.
images = data.test.images[0:9]

# Get the true classes for those images.
cls_true = data.test.cls[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true)
```



TensorFlow图

TensorFlow的全部目的就是使用一个称之为计算图（computational graph）的东西，它会比直接在Python中进行相同计算量要高效得多。TensorFlow比Numpy更高效，因为TensorFlow了解整个需要运行的计算图，然而Numpy只知道某个时间点上唯一的数学运算。

TensorFlow也能够自动地计算需要优化的变量的梯度，使得模型有更好的表现。这是由于图是简单数学表达式的结合，因此整个图的梯度可以用链式法则推导出来。

TensorFlow还能利用多核CPU和GPU，Google也为TensorFlow制造了称为TPUs（Tensor Processing Units）的特殊芯片，它比GPU更快。

一个TensorFlow图由下面几个部分组成，后面会详细描述：

- 占位符变量（Placeholder）用来改变图的输入。
- 模型变量（Model）将会被优化，使得模型表现得更好。
- 模型本质上就是一些数学函数，它根据Placeholder和模型的输入变量来计算一些输出。
- 一个cost度量用来指导变量的优化。
- 一个优化策略会更新模型的变量。

另外，TensorFlow图也包含了一些调试状态，比如用TensorBoard打印log数据，本教程不涉及这些。

占位符 （Placeholder）变量

Placeholder是作为图的输入，我们每次运行图的时候都可能改变它们。将这个过程称为feeding placeholder变量，后面将会描述这个。

首先我们为输入图像定义placeholder变量。这让我们可以改变输入到TensorFlow图中的图像。这也是一个张量（tensor），代表一个多维向量或矩阵。数据类型设置为float32，形状设为 `[None, img_size_flat]` ， `None` 代表tensor可能保存着任意数量的图像，每张图象是一个长度为 `img_size_flat` 的向量。

```
x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name=
'x')
```

卷积层希望 `x` 被编码为4维张量，因此我们需要将它的形状转换至 `[num_images, img_height, img_width, num_channels]` 。注意 `img_height == img_width == img_size` ，如果第一维的大小设为-1， `num_images` 的大小也会被自动推导出来。转换运算如下：

```
x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
```

接下来我们为输入变量 `x` 中的图像所对应的真实标签定义placeholder变量。变量的形状是 `[None, num_classes]` ，这代表着它保存了任意数量的标签，每个标签是长度为 `num_classes` 的向量，本例中长度为10。

```
y_true = tf.placeholder(tf.float32, shape=[None, 10], name='y_tr
ue')
```

我们也可以为class-number提供一个placeholder，但这里用argmax来计算它。这里只是TensorFlow中的一些操作，没有执行什么运算。

```
y_true_cls = tf.argmax(y_true, dimension=1)
```

神经网络

这一节用PrettyTensor实现卷积神经网络，这要比直接在TensorFlow中实现来得简单，详见教程 #03。

基本思想就是用一个Pretty Tensor object封装输入张量 `x_image` ，它有一个添加新卷积层的帮助函数，以此来创建整个神经网络。Pretty Tensor负责变量分配等等。

```
x_pretty = pt.wrap(x_image)
```

现在我们已经将输入图像装到一个PrettyTensor的object中，再用几行代码就可以添加卷积层和全连接层。

注意，在 `with` 代码块中， `pt.defaults_scope(activation_fn=tf.nn.relu)` 把 `activation_fn=tf.nn.relu` 当作每个的层参数，因此这些层都用到了Rectified Linear Units (ReLU) 。 `defaults_scope` 使我们能更方便地修改所有层的参数。

```
with pt.defaults_scope(activation_fn=tf.nn.relu):
    y_pred, loss = x_pretty.\
        conv2d(kernel=5, depth=16, name='layer_conv1').\
        max_pool(kernel=2, stride=2).\
        conv2d(kernel=5, depth=36, name='layer_conv2').\
        max_pool(kernel=2, stride=2).\
        flatten().\
        fully_connected(size=128, name='layer_fc1').\
        softmax_classifier(num_classes=num_classes, labels=y_tru
e)
```

获取权重

下面，我们要绘制神经网络的权重。当使用Pretty Tensor来创建网络时，层的所有变量都是由Pretty Tensoe间接创建的。因此我们要从TensorFlow中获取变量。

我们用 `layer_conv1` 和 `layer_conv2` 代表两个卷积层。这也叫变量作用域（不要与上面描述的 `defaults_scope` 混淆了）。PrettyTensor会自动给它为每个层创建的变量命名，因此我们可以通过层的作用域名称和变量名来取得某一层的权重。

函数实现有点笨拙，因为我们不得不用TensorFlow函数 `get_variable()` ，它是设计给其他用途的，创建新的变量或重用现有变量。创建下面的帮助函数很简单。

```
def get_weights_variable(layer_name):
    # Retrieve an existing variable named 'weights' in the scope
    # with the given layer_name.
    # This is awkward because the TensorFlow function was
    # really intended for another purpose.

    with tf.variable_scope(layer_name, reuse=True):
        variable = tf.get_variable('weights')

    return variable
```

借助这个帮助函数我们可以获取变量。这些是TensorFlow的objects。你需要类似的操作来获取变量的内容： `contents = session.run(weights_conv1)` ，下面会提到这个。

```
weights_conv1 = get_weights_variable(layer_name='layer_conv1')
weights_conv2 = get_weights_variable(layer_name='layer_conv2')
```

优化方法

PrettyTensor给我们提供了预测类型标签(`y_pred`)以及一个需要最小化的损失度量，用来提升神经网络分类图片的能力。

PrettyTensor的文档并没有说明它的损失度量是用cross-entropy还是其他的。但现在我们用 `AdamOptimizer` 来最小化损失。

优化过程并不是在这里执行。实际上，还没计算任何东西，我们只是往TensorFlow图中添加了优化器，以便后续操作。

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(
loss)
```

性能度量

我们需要另外一些性能度量，来向用户展示这个过程。

首先我们从神经网络输出的 `y_pred` 中计算出预测的类别，它是一个包含10个元素的向量。类别数字是最大元素的索引。

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

然后创建一个布尔向量，用来告诉我们每张图片的真实类别是否与预测类别相同。

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

上面的计算先将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确度。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
))
```

Saver

为了保存神经网络的变量，我们创建一个称为Saver-object的对象，它用来保存及恢复TensorFlow图的所有变量。在这里并未保存什么东西，（保存操作）在后面的 `optimize()` 函数中完成。

```
saver = tf.train.Saver()
```

由于（保存操作）常间隔着写在（代码）中，因此保存的文件通常称为checkpoints。

这是用来保存或恢复数据的文件夹。

```
save_dir = 'checkpoints/'
```

如果文件夹不存在则创建。

```
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
```

这是保存checkpoint文件的路径。

```
save_path = os.path.join(save_dir, 'best_validation')
```

运行TensorFlow

创建TensorFlow会话（session）

一旦创建了TensorFlow图，我们需要创建一个TensorFlow会话，用来运行图。

```
session = tf.Session()
```

初始化变量

变量 `weights` 和 `biases` 在优化之前需要先进行初始化。我们写一个简单的封装函数，后面会再次调用。

```
def init_variables():
    session.run(tf.global_variables_initializer())
```

运行函数来初始化变量。

```
init_variables()
```

用来优化迭代的帮助函数

在训练集中有50,000张图。用这些图像计算模型的梯度会花很多时间。因此我们利用随机梯度下降的方法，它在优化器的每次迭代里只用到了一小部分的图像。

如果内存耗尽导致电脑死机或变得很慢，你应该试着减少这些数量，但同时可能还需要更优化的迭代。

```
train_batch_size = 64
```

每迭代100次下面的优化函数，会计算一次验证集上的分类准确率。如果过了1000次迭代验证准确率还是没有提升，就停止优化。我们需要一些变量来跟踪这个过程。

```
# Best validation accuracy seen so far.
best_validation_accuracy = 0.0

# Iteration-number for last improvement to validation accuracy.
last_improvement = 0

# Stop optimization if no improvement found in this many iterati
ons.
require_improvement = 1000
```

函数用来执行一定数量的优化迭代，以此来逐渐改善网络层的变量。在每次迭代中，会从训练集中选择新的一批数据，然后TensorFlow在这些训练样本上执行优化。每100次迭代会打印出（信息），同时计算验证准确率，如果效果有提升的话会将它保存至文件。

```
# Counter for total number of iterations performed so far.
total_iterations = 0

def optimize(num_iterations):
    # Ensure we update the global variables rather than local co
pies.
    global total_iterations
    global best_validation_accuracy
    global last_improvement

    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(num_iterations):

        # Increase the total number of iterations performed.
        # It is easier to update it in each iteration because
        # we need this number several times in the following.
        total_iterations += 1

        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = data.train.next_batch(train_batc
```

```
h_size)

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimiz
er.
        session.run(optimizer, feed_dict=feed_dict_train)

        # Print status every 100 iterations and after last itera
tion.
        if (total_iterations % 100 == 0) or (i == (num_iteration
s - 1)):

            # Calculate the accuracy on the training-batch.
            acc_train = session.run(accuracy, feed_dict=feed_dic
t_train)

            # Calculate the accuracy on the validation-set.
            # The function returns 2 values but we only need the
 first.
            acc_validation, _ = validation_accuracy()

            # If validation accuracy is an improvement over best
-known.
            if acc_validation > best_validation_accuracy:
                # Update the best-known validation accuracy.
                best_validation_accuracy = acc_validation

                # Set the iteration for the last improvement to
current.
                last_improvement = total_iterations

                # Save all variables of the TensorFlow graph to
file.
                saver.save(sess=session, save_path=save_path)

                # A string to be printed below, shows improvemen
t found.
                improved_str = '*'
            else:
                # An empty string to be printed below.
                # Shows that no improvement was found.
                improved_str = ''

            # Status-message for printing.
            msg = "Iter: {0:>6}, Train-Batch Accuracy: {1:>6.1%}
, Validation Acc: {2:>6.1%} {3}"
```

```
            # Print it.
            print(msg.format(i + 1, acc_train, acc_validation, i
mproved_str))

        # If no improvement found in the required number of iter
ations.
        if total_iterations - last_improvement > require_improve
ment:
            print("No improvement found in a while, stopping opt
imization.")

            # Break out from the for-loop.
            break

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_
dif)))))
```

用来绘制错误样本的帮助函数

函数用来绘制测试集中被误分类的样本。

```python
def plot_example_errors(cls_pred, correct):
    # This function is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # correct is a boolean array whether the predicted class
    # is equal to the true class for each image in the test-set.

    # Negate the boolean array.
    incorrect = (correct == False)

    # Get the images from the test-set that have been
    # incorrectly classified.
    images = data.test.images[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = data.test.cls[incorrect]

    # Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])
```

绘制混淆（confusion）矩阵的帮助函数

```
def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the true classifications for the test-set.
    cls_true = data.test.cls

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_true,
                          y_pred=cls_pred)

    # Print the confusion matrix as text.
    print(cm)

    # Plot the confusion matrix as an image.
    plt.matshow(cm)

    # Make various adjustments to the plot.
    plt.colorbar()
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
    plt.yticks(tick_marks, range(num_classes))
    plt.xlabel('Predicted')
    plt.ylabel('True')

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

计算分类的帮助函数

这个函数用来计算图像的预测类别，同时返回一个代表每张图像分类是否正确的布尔数组。

由于计算可能会耗费太多内存，就分批处理。如果你的电脑死机了，试着降低batch-size。

```
# Split the data-set in batches of this size to limit RAM usage.
batch_size = 256

def predict_cls(images, labels, cls_true):
    # Number of images.
    num_images = len(images)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_images, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing thi
s.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        # Create a feed-dict with the images and labels
        # between index i and j.
        feed_dict = {x: images[i:j, :],
                     y_true: labels[i:j, :]}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_d
ict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Create a boolean array whether each image is correctly cla
ssified.
    correct = (cls_true == cls_pred)

    return correct, cls_pred
```

计算测试集上的预测类别。

```
def predict_cls_test():
    return predict_cls(images = data.test.images,
                       labels = data.test.labels,
                       cls_true = data.test.cls)
```

计算验证集上的预测类别。

```
def predict_cls_validation():
    return predict_cls(images = data.validation.images,
                       labels = data.validation.labels,
                       cls_true = data.validation.cls)
```

分类准确率的帮助函数

这个函数计算了给定布尔数组的分类准确率，布尔数组表示每张图像是否被正确分类。比如，
`cls_accuracy([True, True, False, False, False]) = 2/5 = 0.4` 。

```
def cls_accuracy(correct):
    # Calculate the number of correctly classified images.
    # When summing a boolean array, False means 0 and True means
 1.
    correct_sum = correct.sum()

    # Classification accuracy is the number of correctly classif
ied
    # images divided by the total number of images in the test-s
et.
    acc = float(correct_sum) / len(correct)

    return acc, correct_sum
```

计算验证集上的分类准确率。

```
def validation_accuracy():
    # Get the array of booleans whether the classifications are
correct
    # for the validation-set.
    # The function returns two values but we only need the first.

    correct, _ = predict_cls_validation()

    # Calculate the classification accuracy and return it.
    return cls_accuracy(correct)
```

展示性能的帮助函数

函数用来打印测试集上的分类准确率。

为测试集上的所有图片计算分类会花费一段时间，因此我们直接从这个函数里调用上面的函数，这样就不用每个函数都重新计算分类。

```
def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # For all the images in the test-set,
    # calculate the predicted classes and whether they are corre
ct.
    correct, cls_pred = predict_cls_test()

    # Classification accuracy and the number of correct classifi
cations.
    acc, num_correct = cls_accuracy(correct)

    # Number of images being classified.
    num_images = len(correct)

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, num_correct, num_images))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

绘制卷积权重的帮助函数

```
def plot_conv_weights(weights, input_channel=0):
    # Assume weights are TensorFlow ops for 4-dim variables
    # e.g. weights_conv1 or weights_conv2.

    # Retrieve the values of the weight-variables from TensorFlo
w.
    # A feed-dict is not necessary because nothing is calculated.

    w = session.run(weights)

    # Print mean and standard deviation.
    print("Mean: {0:.5f}, Stdev: {1:.5f}".format(w.mean(), w.std
()))

    # Get the lowest and highest values for the weights.
    # This is used to correct the colour intensity across
    # the images so they can be compared with each other.
    w_min = np.min(w)
    w_max = np.max(w)
```

```
    # Number of filters used in the conv. layer.
    num_filters = w.shape[3]

    # Number of grids to plot.
    # Rounded-up, square-root of the number of filters.
    num_grids = math.ceil(math.sqrt(num_filters))

    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(num_grids, num_grids)

    # Plot all the filter-weights.
    for i, ax in enumerate(axes.flat):
        # Only plot the valid filter-weights.
        if i<num_filters:
            # Get the weights for the i'th filter of the input c
hannel.
            # The format of this 4-dim tensor is determined by t
he
            # TensorFlow API. See Tutorial #02 for more details.
            img = w[:, :, input_channel, i]

            # Plot image.
            ax.imshow(img, vmin=w_min, vmax=w_max,
                      interpolation='nearest', cmap='seismic')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

优化之前的性能

测试集上的准确度很低，这是由于模型只做了初始化，并没做任何优化，所以它只是对图像做随机分类。

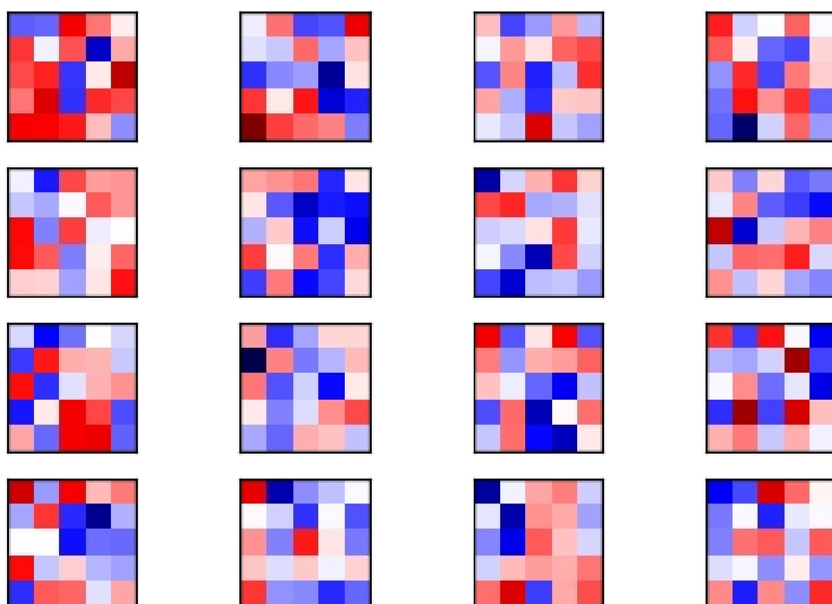
```
print_test_accuracy()
```

```
Accuracy on Test-Set: 8.5% (849 / 10000)
```

卷积权重是随机的，但也很难把它与下面优化过的权重区分开来。这里也展示了平均值和标准差，因此我们可以看看是否有差别。

```
plot_conv_weights(weights=weights_conv1)
```

```
Mean: 0.00880, Stdev: 0.28635
```



10,000次优化迭代后的性能

现在我们进行了10,000次优化迭代，并且，当经过1000次迭代验证集上的性能却没有提升时就停止优化。

星号 * 代表验证集上的分类准确度有提升。

```
optimize(num_iterations=10000)
```

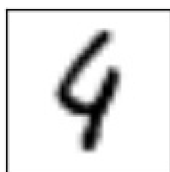
```
Iter:    100, Train-Batch Accuracy:  84.4%, Validation Acc:  85.
2% *
Iter:    200, Train-Batch Accuracy:  92.2%, Validation Acc:  91.
5% *
Iter:    300, Train-Batch Accuracy:  95.3%, Validation Acc:  93.
7% *
Iter:    400, Train-Batch Accuracy:  92.2%, Validation Acc:  94.
3% *
Iter:    500, Train-Batch Accuracy:  98.4%, Validation Acc:  94.
7% *
Iter:    600, Train-Batch Accuracy:  93.8%, Validation Acc:  94.
7%
Iter:    700, Train-Batch Accuracy:  98.4%, Validation Acc:  95.
6% *
```

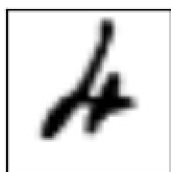
```
Iter:    800, Train-Batch Accuracy: 100.0%, Validation Acc:  96.
3% *
Iter:    900, Train-Batch Accuracy:  98.4%, Validation Acc:  96.
4% *
Iter:   1000, Train-Batch Accuracy: 100.0%, Validation Acc:  96.
9% *
Iter:   1100, Train-Batch Accuracy:  96.9%, Validation Acc:  97.
0% *
Iter:   1200, Train-Batch Accuracy:  93.8%, Validation Acc:  97.
0% *
Iter:   1300, Train-Batch Accuracy:  92.2%, Validation Acc:  97.
2% *
Iter:   1400, Train-Batch Accuracy: 100.0%, Validation Acc:  97.
3% *
Iter:   1500, Train-Batch Accuracy:  96.9%, Validation Acc:  97.
4% *
Iter:   1600, Train-Batch Accuracy: 100.0%, Validation Acc:  97.
7% *
Iter:   1700, Train-Batch Accuracy: 100.0%, Validation Acc:  97.
8% *
Iter:   1800, Train-Batch Accuracy:  98.4%, Validation Acc:  97.
7%
Iter:   1900, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
1% *
Iter:   2000, Train-Batch Accuracy:  95.3%, Validation Acc:  98.
0%
Iter:   2100, Train-Batch Accuracy:  98.4%, Validation Acc:  97.
9%
Iter:   2200, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
0%
Iter:   2300, Train-Batch Accuracy:  96.9%, Validation Acc:  98.
1%
Iter:   2400, Train-Batch Accuracy:  93.8%, Validation Acc:  98.
1%
Iter:   2500, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
2% *
Iter:   2600, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
0%
Iter:   2700, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
0%
Iter:   2800, Train-Batch Accuracy:  96.9%, Validation Acc:  98.
1%
Iter:   2900, Train-Batch Accuracy:  96.9%, Validation Acc:  98.
2%
Iter:   3000, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
2%
Iter:   3100, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
1%
Iter:   3200, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
3% *
Iter:   3300, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
4% *
Iter:   3400, Train-Batch Accuracy:  95.3%, Validation Acc:  98.
```

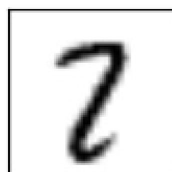
```
0%
Iter:   3500, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
3%
Iter:   3600, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
5% *
Iter:   3700, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
3%
Iter:   3800, Train-Batch Accuracy:  96.9%, Validation Acc:  98.
1%
Iter:   3900, Train-Batch Accuracy:  96.9%, Validation Acc:  98.
5%
Iter:   4000, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
4%
Iter:   4100, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
5%
Iter:   4200, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
3%
Iter:   4300, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
6% *
Iter:   4400, Train-Batch Accuracy:  96.9%, Validation Acc:  98.
4%
Iter:   4500, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
5%
Iter:   4600, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
5%
Iter:   4700, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
4%
Iter:   4800, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
8% *
Iter:   4900, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
8%
Iter:   5000, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
6%
Iter:   5100, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
6%
Iter:   5200, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
6%
Iter:   5300, Train-Batch Accuracy:  96.9%, Validation Acc:  98.
5%
Iter:   5400, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
7%
Iter:   5500, Train-Batch Accuracy:  98.4%, Validation Acc:  98.
6%
Iter:   5600, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
4%
Iter:   5700, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
6%
Iter:   5800, Train-Batch Accuracy: 100.0%, Validation Acc:  98.
7%
No improvement found in a while, stopping optimization.
Time usage: 0:00:28
```

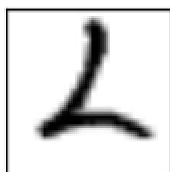
```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

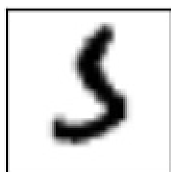
```
Accuracy on Test-Set: 98.4% (9842 / 10000)
Example errors:
```


True: 4, Pred: 9

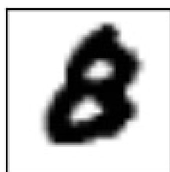

True: 4, Pred: 6

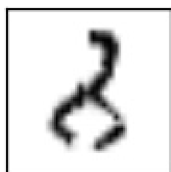

True: 2, Pred: 7


True: 2, Pred: 1


True: 5, Pred: 3

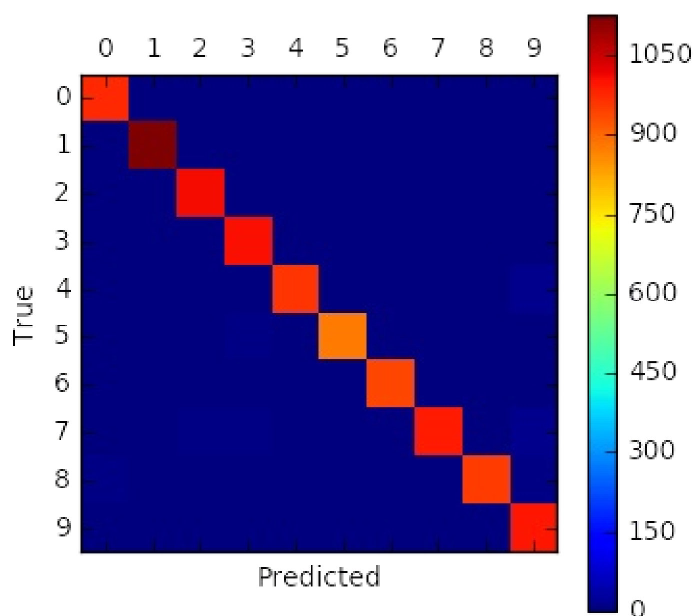

True: 6, Pred: 0


True: 8, Pred: 0


True: 8, Pred: 2


True: 2, Pred: 7

```
Confusion Matrix:
[[ 974    0    0    0    0    1    2    0    2    1]
 [    0 1127    2    2    0    0    1    0    3    0]
 [    4    4 1012    4    1    0    0    3    4    0]
 [    0    0    1 1005    0    2    0    0    2    0]
 [    1    0    1    0  961    0    2    0    3   14]
 [    2    0    1    6    0  880    1    0    1    1]
 [    4    2    0    1    3    4  942    0    2    0]
 [    1    1    8    6    1    0    0  994    1   16]
 [    6    0    1    4    1    1    1    2  952    6]
 [    3    3    0    3    2    2    0    0    1  995]]
```



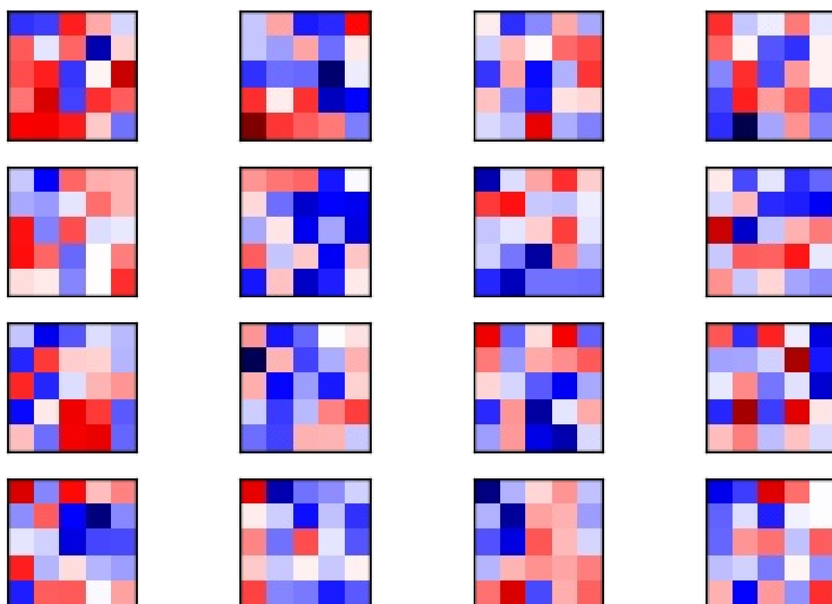
现在卷积权重是经过优化的。将这些与上面的随机权重进行对比。它们看起来基本相同。实际上，一开始我以为程序有bug，因为优化前后的权重看起来差不多。

但保存图像，并排着比较它们（你可以右键保存）。你会发现两者有细微的不同。

平均值和标准差也有一点变化，因此优化过的权重肯定是不一样的。

```
plot_conv_weights(weights=weights_conv1)
```

```
Mean: 0.02895, Stdev: 0.29949
```



再次初始化变量

再一次用随机值来初始化所有神经网络变量。

```
init_variables()
```

这意味着神经网络又是完全随机地对图片进行分类，由于只是随机的猜测所以分类准确率很低。

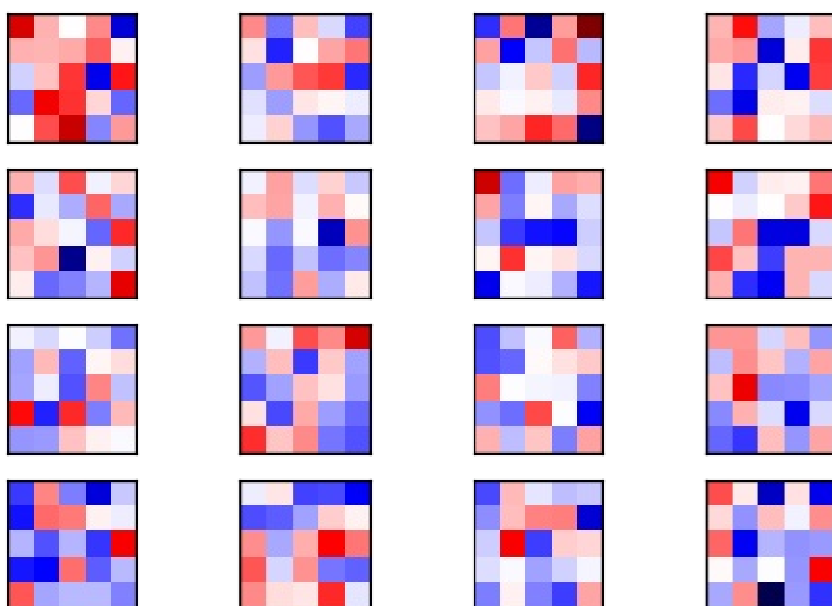
```
print_test_accuracy()
```

```
Accuracy on Test-Set: 13.4% (1341 / 10000)
```

卷积权重看起来应该与上面的不同。

```
plot_conv_weights(weights=weights_conv1)
```

```
Mean: -0.01086, Stdev: 0.28023
```



恢复最好的变量

重新载入在优化过程中保存到文件的所有变量。

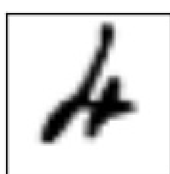
```
saver.restore(sess=session, save_path=save_path)
```

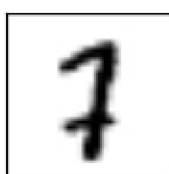
使用之前保存的那些变量，分类准确率又提高了。

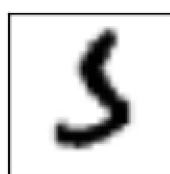
注意，准确率与之前相比可能会有细微的上升或下降，这是由于文件里的变量是用来最大化验证集上的分类准确率，但在保存文件之后，又进行了1000次的优化迭代，因此这是两组有轻微不同的变量的结果。有时这会导致测试集上更好或更差的表现。

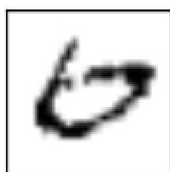
```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

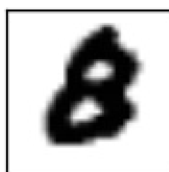
```
Accuracy on Test-Set: 98.3% (9826 / 10000)
Example errors:
```

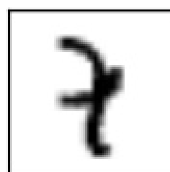

True: 4, Pred: 2

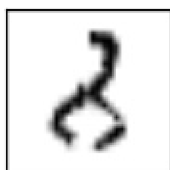

True: 7, Pred: 3


True: 5, Pred: 3

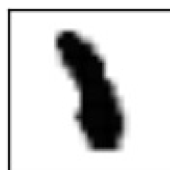

True: 6, Pred: 0


True: 8, Pred: 0

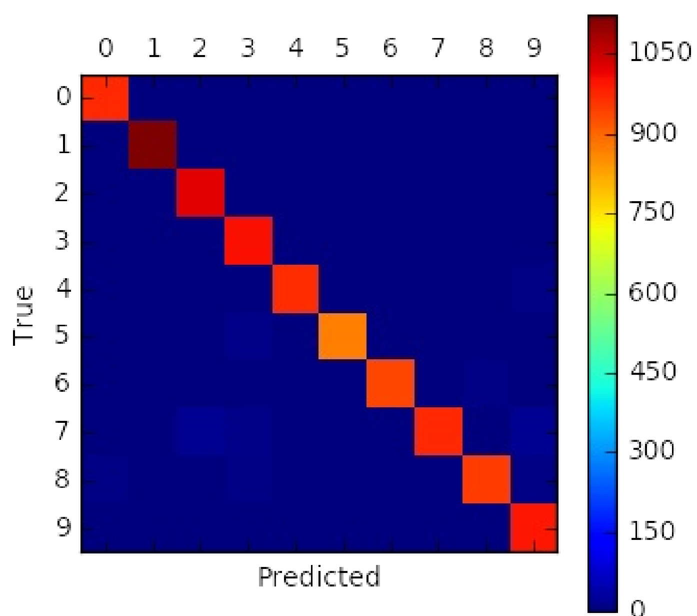

True: 7, Pred: 3


True: 8, Pred: 2


True: 7, Pred: 9


True: 1, Pred: 8

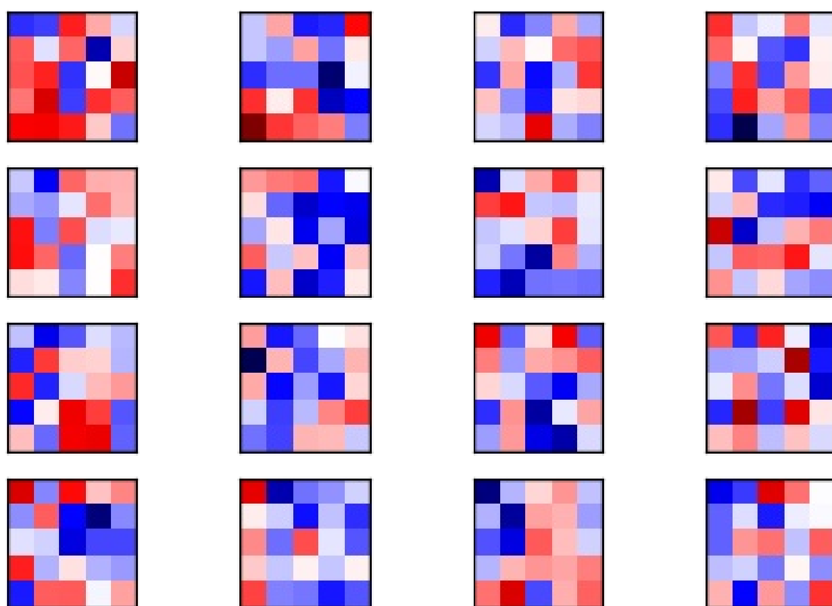
```
Confusion Matrix:
[[ 973    0    0    0    0    0    2    0    3    2]
 [    0 1124    2    2    0    0    3    0    4    0]
 [    2    1 1027    0    0    0    0    1    1    0]
 [    0    0    1 1005    0    2    0    0    2    0]
 [    0    0    3    0  968    0    1    0    3    7]
 [    2    0    1    9    0  871    3    0    3    3]
 [    4    2    1    0    3    3  939    0    6    0]
 [    1    3   19   11    2    0    0  972    2   18]
 [    6    0    3    5    1    0    1    2  951    5]
 [    3    3    0    1    4    1    0    0    1  996]]
```



卷积权重也与之前显示的图几乎相同，同样，由于多做了1000次优化迭代，二者并非完全一样。

```
plot_conv_weights(weights=weights_conv1)
```

```
Mean: 0.02792, Stdev: 0.29822
```



关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# session.close()
```

总结

这篇教程描述了在TensorFlow中如何保存并恢复神经网络的变量。它有许多用处。比如，当你用神经网络来识别图像的时候，只需要训练网络一次，然后可以在其他电脑上完成开发工作。

checkpoint的另一个用处是，如果你有一个非常大的神经网络和数据集，就可能会在中间保存一些checkpoints来避免电脑死机，这样，你就可以在最近的checkpoint开始优化而不是重头开始。

本教程也展示了如何用验证集来进行所谓的Early Stopping，如果没有降低验证错误优化就会终止。这在神经网络出现过拟合以及开始学习训练集中的噪声时很有用；不过这在本教程的神经网络和MNIST数据集中并不是什么大问题。

还有一个有趣的现象，最优化时卷积权重（或者叫滤波）的变化很小，即使网络的性能从随机猜测提高到近乎完美的分类。奇怪的是随机的权重好像已经足够好了。你认为为什么会有这种现象?

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 在经过1000次迭代而性能没有提升时，优化就终止了。这样够吗？你能想出一个更好地进行Early Stopping的方法么？试着实现它。
- 如果checkpoint文件已经存在了，载入它而不是做优化。
- 每100次优化迭代保存一次checkpoint。通过 `saver.latest_checkpoint()` 取回最新的（保存点）。为什么保存多个checkpoints而不是只保存最近的一个？
- 试着改变神经网络，比如添加其他层。当你从不同的网络中重新载入变量会出现什么问题？
- 用 `plot_conv_weights()` 函数在优化前后画出第二个卷积层的权重。它们几乎相同的么？
- 你认为优化过的卷积权重为什么与随机初始化的（权重）几乎相同？
- 不看源码，自己重写程序。
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #05

集成学习

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

简介

这篇教程介绍了卷积神经网络的集成（ensemble）。我们使用多个神经网络，然后取它们输出的平均，而不是只用一个。

最终也是在MINIST数据集上识别手写数字。ensemble稍微地提升了测试集上的分类准确率，但差异很小，也可能是随机出现的。此外，ensemble误分类的一些图像在单独网络上却是正确分类的。

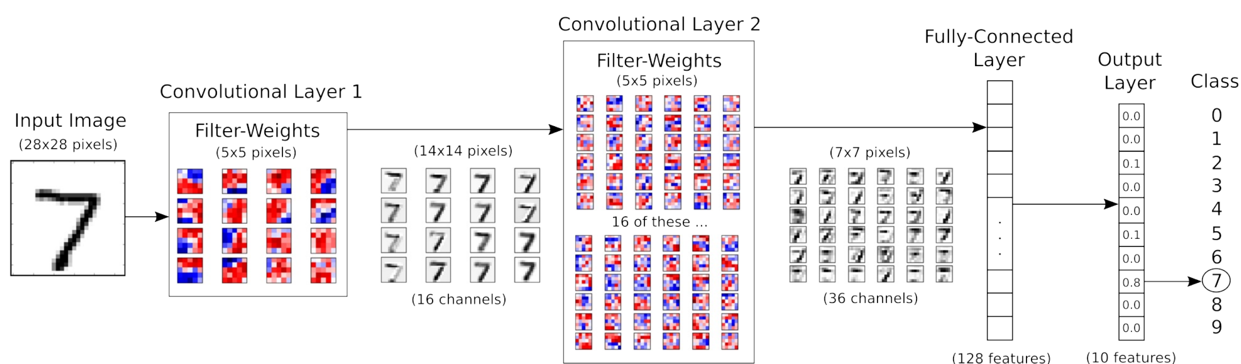
本文基于上一篇教程，你需要了解基本的TensorFlow和附加包Pretty Tensor。其中大量代码和文字与之前教程相似，如果你已经看过就可以快速地浏览本文。

流程图

下面的图表直接显示了之后实现的卷积神经网络中数据的传递。网络有两个卷积层和两个全连接层，最后一层是用来给输入图像分类的。关于网络和卷积的更多细节描述见教程 #02 。

本教程实现了5个这样的神经网络的集成，每个网络的结构相同但权重以及其他变量不同。

```
from IPython.display import Image
Image('images/02_network_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
import time
from datetime import timedelta
import math
import os

# Use PrettyTensor to simplify Neural Network construction.
import prettytensor as pt
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.12.0-rc0'
```

PrettyTensor 版本:

```
pt.__version__
```

```
'0.7.1'
```

载入数据

MNIST数据集大约12MB，如果没在给定路径中找到就会自动下载。

```
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets('data/MNIST/', one_hot=True)
```

```
Extracting data/MNIST/train-images-idx3-ubyte.gz
Extracting data/MNIST/train-labels-idx1-ubyte.gz
Extracting data/MNIST/t10k-images-idx3-ubyte.gz
Extracting data/MNIST/t10k-labels-idx1-ubyte.gz
```

现在已经载入了MNIST数据集，它由70,000张图像和对应的标签（比如图像的类别）组成。数据集分成三份互相独立的子集，但后面我们会生成随机的训练集。

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(data.train.labels)))
print("- Test-set:\t\t{}".format(len(data.test.labels)))
print("- Validation-set:\t{}".format(len(data.validation.labels)
))
```

```
Size of:
- Training-set:		55000
- Test-set:		10000
- Validation-set:	5000
```

类别数字

类型标签使用One-Hot编码，这意外每个标签是长为10的向量，除了一个元素之外，其他的都为零。这个元素的索引就是类别的数字，即相应图片中画的数字。我们也需要测试集和验证集的整形类别数字，在这里计算。

```
data.test.cls = np.argmax(data.test.labels, axis=1)
data.validation.cls = np.argmax(data.validation.labels, axis=1)
```

创建随机训练集的帮助函数

我们将会在随机选择的训练集上训练5个不同的神经网络。首先，将原始训练集和验证集合并到大的一个数组中。图像和标签都要进行此操作。

```
combined_images = np.concatenate([data.train.images, data.valida
tion.images], axis=0)
combined_labels = np.concatenate([data.train.labels, data.valida
tion.labels], axis=0)
```

检查合并后的数组大小是否正确。

```
print(combined_images.shape)
print(combined_labels.shape)
```

```
(60000, 784)
(60000, 10)
```

合并数据集的大小。

```
combined_size = len(combined_images)
combined_size
```

```
60000
```

定义每个神经网络使用的训练集的大小。你可以试着改变大小。

```
train_size = int(0.8 * combined_size)
train_size
```

```
48000
```

在训练时并没有使用验证集，但它的大小如下。

```
validation_size = combined_size - train_size
validation_size
```

```
12000
```

帮助函数将合并数组集划分成随机的训练集和验证集。

```
def random_training_set():
    # Create a randomized index into the full / combined trainin
g-set.
    idx = np.random.permutation(combined_size)

    # Split the random index into training- and validation-sets.
    idx_train = idx[0:train_size]
    idx_validation = idx[train_size:]

    # Select the images and labels for the new training-set.
    x_train = combined_images[idx_train, :]
    y_train = combined_labels[idx_train, :]

    # Select the images and labels for the new validation-set.
    x_validation = combined_images[idx_validation, :]
    y_validation = combined_labels[idx_validation, :]

    # Return the new training- and validation-sets.
    return x_train, y_train, x_validation, y_validation
```

数据维度

在下面的源码中，有很多地方用到了数据维度。它们只在一个地方定义，因此我们可以在代码中使用这些变量而不是直接写数字。

```
# We know that MNIST images are 28 pixels in each dimension.
img_size = 28

# Images are stored in one-dimensional arrays of this length.
img_size_flat = img_size * img_size

# Tuple with height and width of images used to reshape arrays.
img_shape = (img_size, img_size)

# Number of colour channels for the images: 1 channel for gray-s
cale.
num_channels = 1

# Number of classes, one class for each of 10 digits.
num_classes = 10
```

用来绘制图片的帮助函数

这个函数用来在3x3的栅格中画9张图像，然后在每张图像下面写出真实类别和预测类别。

```
def plot_images(images,                       # Images to plot, 2-d array.
                cls_true,                     # True class-no for images.
                ensemble_cls_pred=None,       # Ensemble predicted class-no.
                best_cls_pred=None):          # Best-net predicted class-no.

    assert len(images) == len(cls_true)

    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(3, 3)

    # Adjust vertical spacing if we need to print ensemble and best-net.
    if ensemble_cls_pred is None:
        hspace = 0.3
    else:
        hspace = 1.0
    fig.subplots_adjust(hspace=hspace, wspace=0.3)

    # For each of the sub-plots.
    for i, ax in enumerate(axes.flat):

        # There may not be enough images for all sub-plots.
        if i < len(images):
            # Plot image.
            ax.imshow(images[i].reshape(img_shape), cmap='binary')

            # Show true and predicted classes.
            if ensemble_cls_pred is None:
                xlabel = "True: {0}".format(cls_true[i])
            else:
                msg = "True: {0}\nEnsemble: {1}\nBest Net: {2}"
                xlabel = msg.format(cls_true[i],
                                    ensemble_cls_pred[i],
                                    best_cls_pred[i])

            # Show the classes as the label on the x-axis.
            ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

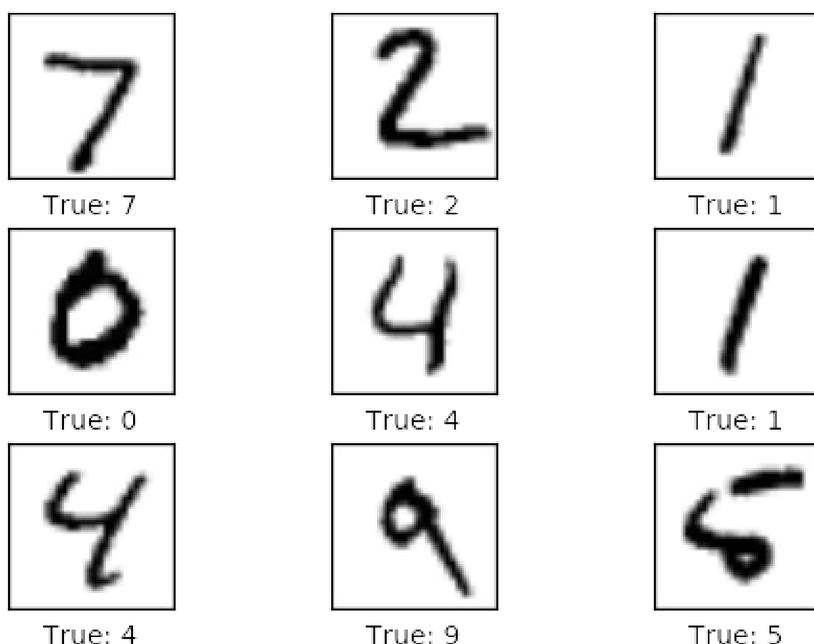
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

绘制几张图像来看看数据是否正确

```
# Get the first images from the test-set.
images = data.test.images[0:9]

# Get the true classes for those images.
cls_true = data.test.cls[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true)
```



TensorFlow图

TensorFlow的全部目的就是使用一个称之为计算图（computational graph）的东西，它会比直接在Python中进行相同计算量要高效得多。TensorFlow比Numpy更高效，因为TensorFlow了解整个需要运行的计算图，然而Numpy只知道某个时间点上唯一的数学运算。

TensorFlow也能够自动地计算需要优化的变量的梯度，使得模型有更好的表现。这是由于图是简单数学表达式的结合，因此整个图的梯度可以用链式法则推导出来。

TensorFlow还能利用多核CPU和GPU，Google也为TensorFlow制造了称为TPUs（Tensor Processing Units）的特殊芯片，它比GPU更快。

一个TensorFlow图由下面几个部分组成，后面会详细描述：

- 占位符变量（Placeholder）用来改变图的输入。
- 模型变量（Model）将会被优化，使得模型表现得更好。
- 模型本质上就是一些数学函数，它根据Placeholder和模型的输入变量来计算一

些输出。
- 一个cost度量用来指导变量的优化。
- 一个优化策略会更新模型的变量。

另外，TensorFlow图也包含了一些调试状态，比如用TensorBoard打印log数据，本教程不涉及这些。

占位符（Placeholder）变量

Placeholder是作为图的输入，我们每次运行图的时候都可能改变它们。将这个过程称为feeding placeholder变量，后面将会描述这个。

首先我们为输入图像定义placeholder变量。这让我们可以改变输入到TensorFlow图中的图像。这也是一个张量（tensor），代表一个多维向量或矩阵。数据类型设置为float32，形状设为 `[None, img_size_flat]` ， `None` 代表tensor可能保存着任意数量的图像，每张图象是一个长度为 `img_size_flat` 的向量。

```
x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name='x')
```

卷积层希望 `x` 被编码为4维张量，因此我们需要将它的形状转换至 `[num_images, img_height, img_width, num_channels]` 。注意 `img_height == img_width == img_size` ，如果第一维的大小设为-1， `num_images` 的大小也会被自动推导出来。转换运算如下：

```
x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
```

接下来我们为输入变量 `x` 中的图像所对应的真实标签定义placeholder变量。变量的形状是 `[None, num_classes]` ，这代表着它保存了任意数量的标签，每个标签是长度为 `num_classes` 的向量，本例中长度为10。

```
y_true = tf.placeholder(tf.float32, shape=[None, 10], name='y_true')
```

我们也可以为class-number提供一个placeholder，但这里用argmax来计算它。这里只是TensorFlow中的一些操作，没有执行什么运算。

```
y_true_cls = tf.argmax(y_true, dimension=1)
```

神经网络

这一节用PrettyTensor实现卷积神经网络，这要比直接在TensorFlow中实现来得简单，详见教程 #03。

基本思想就是用一个Pretty Tensor object封装输入张量 `x_image` ，它有一个添加新卷积层的帮助函数，以此来创建整个神经网络。Pretty Tensor负责变量分配等等。

```
x_pretty = pt.wrap(x_image)
```

现在我们已经将输入图像装到一个PrettyTensor的object中，再用几行代码就可以添加卷积层和全连接层。

注意，在 `with` 代码块中， `pt.defaults_scope(activation_fn=tf.nn.relu)` 把 `activation_fn=tf.nn.relu` 当作每个的层参数，因此这些层都用到了Rectified Linear Units (ReLU) 。 `defaults_scope` 使我们能更方便地修改所有层的参数。

```
with pt.defaults_scope(activation_fn=tf.nn.relu):
    y_pred, loss = x_pretty.\
        conv2d(kernel=5, depth=16, name='layer_conv1').\
        max_pool(kernel=2, stride=2).\
        conv2d(kernel=5, depth=36, name='layer_conv2').\
        max_pool(kernel=2, stride=2).\
        flatten().\
        fully_connected(size=128, name='layer_fc1').\
        softmax_classifier(num_classes=num_classes, labels=y_tru
e)
```

优化方法

PrettyTensor给我们提供了预测类型标签(`y_pred`)以及一个需要最小化的损失度量，用来提升神经网络分类图片的能力。

PrettyTensor的文档并没有说明它的损失度量是用cross-entropy还是其他的。但现在我们用 `AdamOptimizer` 来最小化损失。

优化过程并不是在这里执行。实际上，还没计算任何东西，我们只是往TensorFlow图中添加了优化器，以便后续操作。

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(
loss)
```

性能度量

我们需要另外一些性能度量，来向用户展示这个过程。

首先我们从神经网络输出的 `y_pred` 中计算出预测的类别，它是一个包含10个元素的向量。类别数字是最大元素的索引。

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

然后创建一个布尔向量，用来告诉我们每张图片的真实类别是否与预测类别相同。

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

上面的计算先将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确度。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
))
```

Saver

为了保存神经网络的变量，我们创建一个称为Saver-object的对象，它用来保存及恢复TensorFlow图的所有变量。在这里并未保存什么东西，（保存操作）在后面的 `optimize()` 函数中完成。

注意，如果在ensemble中有超过100个的神经网络，你需要根据情况来增加 `max_to_keep` 。

```
saver = tf.train.Saver(max_to_keep=100)
```

这是用来保存或恢复数据的文件夹。

```
save_dir = 'checkpoints/'
```

如果文件夹不存在则创建。

```
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
```

这个函数根据输入的网络编号返回数据文件的保存路径。

```
def get_save_path(net_number):
    return save_dir + 'network' + str(net_number)
```

运行TensorFlow

创建TensorFlow会话（session）

一旦创建了TensorFlow图，我们需要创建一个TensorFlow会话，用来运行图。

```
session = tf.Session()
```

初始化变量

变量 `weights` 和 `biases` 在优化之前需要先进行初始化。我们写一个简单的封装函数，后面会再次调用。

```
def init_variables():
    session.run(tf.initialize_all_variables())
```

创建随机训练batch的帮助函数

在训练集中有上千张图。用这些图像计算模型的梯度会花很多时间。因此，它在优化器的每次迭代里只用到了一小部分的图像。

如果内存耗尽导致电脑死机或变得很慢，你应该试着减少这些数量，但同时可能还需要更优化的迭代。

```
train_batch_size = 64
```

函数根据给定的大小挑选一个随机的training-batch。

```
def random_batch(x_train, y_train):
    # Total number of images in the training-set.
    num_images = len(x_train)

    # Create a random index into the training-set.
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)

    # Use the random index to select random images and labels.
    x_batch = x_train[idx, :]  # Images.
    y_batch = y_train[idx, :]  # Labels.

    # Return the batch.
    return x_batch, y_batch
```

执行优化迭代的帮助函数

函数用来执行一定数量的优化迭代，以此来逐渐改善网络层的变量。在每次迭代中，会从训练集中选择新的一批数据，然后TensorFlow在这些训练样本上执行优化。每100次迭代会打印出（信息）。

```
def optimize(num_iterations, x_train, y_train):
    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(num_iterations):

        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = random_batch(x_train, y_train)

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimiz
er.
        session.run(optimizer, feed_dict=feed_dict_train)

        # Print status every 100 iterations and after last itera
tion.
        if i % 100 == 0:

            # Calculate the accuracy on the training-batch.
            acc = session.run(accuracy, feed_dict=feed_dict_trai
n)

            # Status-message for printing.
            msg = "Optimization Iteration: {0:>6}, Training Batc
h Accuracy: {1:>6.1%}"

            # Print it.
            print(msg.format(i + 1, acc))

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_
dif)))))
```

创建神经网络的集成（ensemble）

神经网络ensemble的数量

```
num_networks = 5
```

每个神经网络优化迭代的次数。

```
num_iterations = 10000
```

创建神经网络的ensemble。所有网络都使用上面定义的那个TensorFlow图。每个网络的TensorFlow权重和变量都用随机值初始化，然后进行优化。接着将变量保存到磁盘中以便之后重载使用。

如果你只是想重新运行Notebook来对结果进行不同的分析，可以跳过这一步。

```
if True:
    # For each of the neural networks.
    for i in range(num_networks):
        print("Neural network: {0}".format(i))

        # Create a random training-set. Ignore the validation-se
t.
        x_train, y_train, _, _ = random_training_set()

        # Initialize the variables of the TensorFlow graph.
        session.run(tf.global_variables_initializer())

        # Optimize the variables using this training-set.
        optimize(num_iterations=num_iterations,
                 x_train=x_train,
                 y_train=y_train)

        # Save the optimized variables to disk.
        saver.save(sess=session, save_path=get_save_path(i))

        # Print newline.
        print()
```

```
Neural network: 0
Optimization Iteration:      1, Training Batch Accuracy:   6.2%
Optimization Iteration:    101, Training Batch Accuracy:  87.5%
Optimization Iteration:    201, Training Batch Accuracy:  92.2%
Optimization Iteration:    301, Training Batch Accuracy:  92.2%
Optimization Iteration:    401, Training Batch Accuracy:  98.4%
Optimization Iteration:    501, Training Batch Accuracy:  95.3%
Optimization Iteration:    601, Training Batch Accuracy:  95.3%
Optimization Iteration:    701, Training Batch Accuracy:  96.9%
Optimization Iteration:    801, Training Batch Accuracy:  96.9%
Optimization Iteration:    901, Training Batch Accuracy:  98.4%
Optimization Iteration:   1001, Training Batch Accuracy:  95.3%
```

```
Optimization Iteration:   1101, Training Batch Accuracy:  96.9%
Optimization Iteration:   1201, Training Batch Accuracy:  95.3%
Optimization Iteration:   1301, Training Batch Accuracy: 100.0%
Optimization Iteration:   1401, Training Batch Accuracy:  98.4%
Optimization Iteration:   1501, Training Batch Accuracy: 100.0%
Optimization Iteration:   1601, Training Batch Accuracy:  98.4%
Optimization Iteration:   1701, Training Batch Accuracy:  96.9%
Optimization Iteration:   1801, Training Batch Accuracy: 100.0%
Optimization Iteration:   1901, Training Batch Accuracy:  98.4%
Optimization Iteration:   2001, Training Batch Accuracy:  95.3%
Optimization Iteration:   2101, Training Batch Accuracy:  98.4%
Optimization Iteration:   2201, Training Batch Accuracy:  96.9%
Optimization Iteration:   2301, Training Batch Accuracy:  96.9%
Optimization Iteration:   2401, Training Batch Accuracy: 100.0%
Optimization Iteration:   2501, Training Batch Accuracy:  96.9%
Optimization Iteration:   2601, Training Batch Accuracy: 100.0%
Optimization Iteration:   2701, Training Batch Accuracy:  98.4%
Optimization Iteration:   2801, Training Batch Accuracy:  98.4%
Optimization Iteration:   2901, Training Batch Accuracy:  98.4%
Optimization Iteration:   3001, Training Batch Accuracy:  95.3%
Optimization Iteration:   3101, Training Batch Accuracy: 100.0%
Optimization Iteration:   3201, Training Batch Accuracy:  98.4%
Optimization Iteration:   3301, Training Batch Accuracy:  98.4%
Optimization Iteration:   3401, Training Batch Accuracy:  96.9%
Optimization Iteration:   3501, Training Batch Accuracy:  98.4%
Optimization Iteration:   3601, Training Batch Accuracy: 100.0%
Optimization Iteration:   3701, Training Batch Accuracy:  98.4%
Optimization Iteration:   3801, Training Batch Accuracy: 100.0%
Optimization Iteration:   3901, Training Batch Accuracy: 100.0%
Optimization Iteration:   4001, Training Batch Accuracy: 100.0%
Optimization Iteration:   4101, Training Batch Accuracy: 100.0%
Optimization Iteration:   4201, Training Batch Accuracy:  96.9%
Optimization Iteration:   4301, Training Batch Accuracy:  98.4%
Optimization Iteration:   4401, Training Batch Accuracy: 100.0%
Optimization Iteration:   4501, Training Batch Accuracy:  96.9%
Optimization Iteration:   4601, Training Batch Accuracy: 100.0%
Optimization Iteration:   4701, Training Batch Accuracy:  96.9%
Optimization Iteration:   4801, Training Batch Accuracy:  96.9%
Optimization Iteration:   4901, Training Batch Accuracy: 100.0%
Optimization Iteration:   5001, Training Batch Accuracy:  95.3%
Optimization Iteration:   5101, Training Batch Accuracy: 100.0%
Optimization Iteration:   5201, Training Batch Accuracy:  98.4%
Optimization Iteration:   5301, Training Batch Accuracy: 100.0%
Optimization Iteration:   5401, Training Batch Accuracy: 100.0%
Optimization Iteration:   5501, Training Batch Accuracy: 100.0%
Optimization Iteration:   5601, Training Batch Accuracy: 100.0%
Optimization Iteration:   5701, Training Batch Accuracy: 100.0%
Optimization Iteration:   5801, Training Batch Accuracy: 100.0%
Optimization Iteration:   5901, Training Batch Accuracy: 100.0%
Optimization Iteration:   6001, Training Batch Accuracy: 100.0%
Optimization Iteration:   6101, Training Batch Accuracy: 100.0%
Optimization Iteration:   6201, Training Batch Accuracy: 100.0%
Optimization Iteration:   6301, Training Batch Accuracy:  98.4%
```

```
Optimization Iteration:   6401, Training Batch Accuracy:  98.4%
Optimization Iteration:   6501, Training Batch Accuracy: 100.0%
Optimization Iteration:   6601, Training Batch Accuracy: 100.0%
Optimization Iteration:   6701, Training Batch Accuracy: 100.0%
Optimization Iteration:   6801, Training Batch Accuracy: 100.0%
Optimization Iteration:   6901, Training Batch Accuracy:  98.4%
Optimization Iteration:   7001, Training Batch Accuracy:  98.4%
Optimization Iteration:   7101, Training Batch Accuracy: 100.0%
Optimization Iteration:   7201, Training Batch Accuracy:  98.4%
Optimization Iteration:   7301, Training Batch Accuracy: 100.0%
Optimization Iteration:   7401, Training Batch Accuracy: 100.0%
Optimization Iteration:   7501, Training Batch Accuracy: 100.0%
Optimization Iteration:   7601, Training Batch Accuracy: 100.0%
Optimization Iteration:   7701, Training Batch Accuracy:  98.4%
Optimization Iteration:   7801, Training Batch Accuracy:  96.9%
Optimization Iteration:   7901, Training Batch Accuracy: 100.0%
Optimization Iteration:   8001, Training Batch Accuracy:  98.4%
Optimization Iteration:   8101, Training Batch Accuracy:  98.4%
Optimization Iteration:   8201, Training Batch Accuracy: 100.0%
Optimization Iteration:   8301, Training Batch Accuracy: 100.0%
Optimization Iteration:   8401, Training Batch Accuracy: 100.0%
Optimization Iteration:   8501, Training Batch Accuracy: 100.0%
Optimization Iteration:   8601, Training Batch Accuracy: 100.0%
Optimization Iteration:   8701, Training Batch Accuracy: 100.0%
Optimization Iteration:   8801, Training Batch Accuracy:  96.9%
Optimization Iteration:   8901, Training Batch Accuracy: 100.0%
Optimization Iteration:   9001, Training Batch Accuracy: 100.0%
Optimization Iteration:   9101, Training Batch Accuracy:  98.4%
Optimization Iteration:   9201, Training Batch Accuracy: 100.0%
Optimization Iteration:   9301, Training Batch Accuracy: 100.0%
Optimization Iteration:   9401, Training Batch Accuracy: 100.0%
Optimization Iteration:   9501, Training Batch Accuracy:  98.4%
Optimization Iteration:   9601, Training Batch Accuracy: 100.0%
Optimization Iteration:   9701, Training Batch Accuracy: 100.0%
Optimization Iteration:   9801, Training Batch Accuracy: 100.0%
Optimization Iteration:   9901, Training Batch Accuracy: 100.0%
Time usage: 0:00:40

Neural network: 1
Optimization Iteration:      1, Training Batch Accuracy:   7.8%
Optimization Iteration:    101, Training Batch Accuracy:  85.9%
Optimization Iteration:    201, Training Batch Accuracy:  95.3%
Optimization Iteration:    301, Training Batch Accuracy:  90.6%
Optimization Iteration:    401, Training Batch Accuracy:  92.2%
Optimization Iteration:    501, Training Batch Accuracy:  95.3%
Optimization Iteration:    601, Training Batch Accuracy:  95.3%
Optimization Iteration:    701, Training Batch Accuracy:  93.8%
Optimization Iteration:    801, Training Batch Accuracy:  96.9%
Optimization Iteration:    901, Training Batch Accuracy:  95.3%
Optimization Iteration:   1001, Training Batch Accuracy:  95.3%
Optimization Iteration:   1101, Training Batch Accuracy:  96.9%
Optimization Iteration:   1201, Training Batch Accuracy:  96.9%
Optimization Iteration:   1301, Training Batch Accuracy:  98.4%
```

```
Optimization Iteration:   1401, Training Batch Accuracy: 100.0%
Optimization Iteration:   1501, Training Batch Accuracy: 100.0%
Optimization Iteration:   1601, Training Batch Accuracy:  96.9%
Optimization Iteration:   1701, Training Batch Accuracy:  95.3%
Optimization Iteration:   1801, Training Batch Accuracy:  98.4%
Optimization Iteration:   1901, Training Batch Accuracy:  98.4%
Optimization Iteration:   2001, Training Batch Accuracy:  98.4%
Optimization Iteration:   2101, Training Batch Accuracy:  98.4%
Optimization Iteration:   2201, Training Batch Accuracy: 100.0%
Optimization Iteration:   2301, Training Batch Accuracy:  98.4%
Optimization Iteration:   2401, Training Batch Accuracy:  96.9%
Optimization Iteration:   2501, Training Batch Accuracy: 100.0%
Optimization Iteration:   2601, Training Batch Accuracy:  98.4%
Optimization Iteration:   2701, Training Batch Accuracy:  95.3%
Optimization Iteration:   2801, Training Batch Accuracy:  96.9%
Optimization Iteration:   2901, Training Batch Accuracy:  96.9%
Optimization Iteration:   3001, Training Batch Accuracy:  98.4%
Optimization Iteration:   3101, Training Batch Accuracy: 100.0%
Optimization Iteration:   3201, Training Batch Accuracy: 100.0%
Optimization Iteration:   3301, Training Batch Accuracy:  96.9%
Optimization Iteration:   3401, Training Batch Accuracy:  96.9%
Optimization Iteration:   3501, Training Batch Accuracy:  98.4%
Optimization Iteration:   3601, Training Batch Accuracy:  98.4%
Optimization Iteration:   3701, Training Batch Accuracy: 100.0%
Optimization Iteration:   3801, Training Batch Accuracy:  96.9%
Optimization Iteration:   3901, Training Batch Accuracy: 100.0%
Optimization Iteration:   4001, Training Batch Accuracy: 100.0%
Optimization Iteration:   4101, Training Batch Accuracy: 100.0%
Optimization Iteration:   4201, Training Batch Accuracy: 100.0%
Optimization Iteration:   4301, Training Batch Accuracy:  98.4%
Optimization Iteration:   4401, Training Batch Accuracy: 100.0%
Optimization Iteration:   4501, Training Batch Accuracy:  98.4%
Optimization Iteration:   4601, Training Batch Accuracy:  98.4%
Optimization Iteration:   4701, Training Batch Accuracy:  98.4%
Optimization Iteration:   4801, Training Batch Accuracy:  96.9%
Optimization Iteration:   4901, Training Batch Accuracy:  98.4%
Optimization Iteration:   5001, Training Batch Accuracy:  98.4%
Optimization Iteration:   5101, Training Batch Accuracy: 100.0%
Optimization Iteration:   5201, Training Batch Accuracy: 100.0%
Optimization Iteration:   5301, Training Batch Accuracy: 100.0%
Optimization Iteration:   5401, Training Batch Accuracy: 100.0%
Optimization Iteration:   5501, Training Batch Accuracy:  98.4%
Optimization Iteration:   5601, Training Batch Accuracy: 100.0%
Optimization Iteration:   5701, Training Batch Accuracy:  96.9%
Optimization Iteration:   5801, Training Batch Accuracy:  98.4%
Optimization Iteration:   5901, Training Batch Accuracy:  96.9%
Optimization Iteration:   6001, Training Batch Accuracy: 100.0%
Optimization Iteration:   6101, Training Batch Accuracy: 100.0%
Optimization Iteration:   6201, Training Batch Accuracy: 100.0%
Optimization Iteration:   6301, Training Batch Accuracy:  98.4%
Optimization Iteration:   6401, Training Batch Accuracy:  98.4%
Optimization Iteration:   6501, Training Batch Accuracy: 100.0%
Optimization Iteration:   6601, Training Batch Accuracy: 100.0%
```

```
Optimization Iteration:   6701, Training Batch Accuracy:  98.4%
Optimization Iteration:   6801, Training Batch Accuracy: 100.0%
Optimization Iteration:   6901, Training Batch Accuracy: 100.0%
Optimization Iteration:   7001, Training Batch Accuracy: 100.0%
Optimization Iteration:   7101, Training Batch Accuracy: 100.0%
Optimization Iteration:   7201, Training Batch Accuracy:  96.9%
Optimization Iteration:   7301, Training Batch Accuracy: 100.0%
Optimization Iteration:   7401, Training Batch Accuracy: 100.0%
Optimization Iteration:   7501, Training Batch Accuracy: 100.0%
Optimization Iteration:   7601, Training Batch Accuracy: 100.0%
Optimization Iteration:   7701, Training Batch Accuracy:  98.4%
Optimization Iteration:   7801, Training Batch Accuracy: 100.0%
Optimization Iteration:   7901, Training Batch Accuracy: 100.0%
Optimization Iteration:   8001, Training Batch Accuracy:  98.4%
Optimization Iteration:   8101, Training Batch Accuracy: 100.0%
Optimization Iteration:   8201, Training Batch Accuracy: 100.0%
Optimization Iteration:   8301, Training Batch Accuracy: 100.0%
Optimization Iteration:   8401, Training Batch Accuracy:  98.4%
Optimization Iteration:   8501, Training Batch Accuracy:  98.4%
Optimization Iteration:   8601, Training Batch Accuracy: 100.0%
Optimization Iteration:   8701, Training Batch Accuracy:  98.4%
Optimization Iteration:   8801, Training Batch Accuracy: 100.0%
Optimization Iteration:   8901, Training Batch Accuracy: 100.0%
Optimization Iteration:   9001, Training Batch Accuracy: 100.0%
Optimization Iteration:   9101, Training Batch Accuracy: 100.0%
Optimization Iteration:   9201, Training Batch Accuracy: 100.0%
Optimization Iteration:   9301, Training Batch Accuracy: 100.0%
Optimization Iteration:   9401, Training Batch Accuracy: 100.0%
Optimization Iteration:   9501, Training Batch Accuracy: 100.0%
Optimization Iteration:   9601, Training Batch Accuracy: 100.0%
Optimization Iteration:   9701, Training Batch Accuracy: 100.0%
Optimization Iteration:   9801, Training Batch Accuracy:  98.4%
Optimization Iteration:   9901, Training Batch Accuracy:  98.4%
Time usage: 0:00:40

Neural network: 2
Optimization Iteration:      1, Training Batch Accuracy:   3.1%
Optimization Iteration:    101, Training Batch Accuracy:  84.4%
Optimization Iteration:    201, Training Batch Accuracy:  87.5%
Optimization Iteration:    301, Training Batch Accuracy:  87.5%
Optimization Iteration:    401, Training Batch Accuracy:  98.4%
Optimization Iteration:    501, Training Batch Accuracy:  93.8%
Optimization Iteration:    601, Training Batch Accuracy:  98.4%
Optimization Iteration:    701, Training Batch Accuracy:  93.8%
Optimization Iteration:    801, Training Batch Accuracy: 100.0%
Optimization Iteration:    901, Training Batch Accuracy: 100.0%
Optimization Iteration:   1001, Training Batch Accuracy:  96.9%
Optimization Iteration:   1101, Training Batch Accuracy:  93.8%
Optimization Iteration:   1201, Training Batch Accuracy:  96.9%
Optimization Iteration:   1301, Training Batch Accuracy:  96.9%
Optimization Iteration:   1401, Training Batch Accuracy:  95.3%
Optimization Iteration:   1501, Training Batch Accuracy:  98.4%
Optimization Iteration:   1601, Training Batch Accuracy:  96.9%
```

```
Optimization Iteration:   1701, Training Batch Accuracy:  95.3%
Optimization Iteration:   1801, Training Batch Accuracy:  98.4%
Optimization Iteration:   1901, Training Batch Accuracy: 100.0%
Optimization Iteration:   2001, Training Batch Accuracy:  98.4%
Optimization Iteration:   2101, Training Batch Accuracy:  98.4%
Optimization Iteration:   2201, Training Batch Accuracy:  98.4%
Optimization Iteration:   2301, Training Batch Accuracy:  96.9%
Optimization Iteration:   2401, Training Batch Accuracy:  95.3%
Optimization Iteration:   2501, Training Batch Accuracy:  92.2%
Optimization Iteration:   2601, Training Batch Accuracy: 100.0%
Optimization Iteration:   2701, Training Batch Accuracy: 100.0%
Optimization Iteration:   2801, Training Batch Accuracy:  95.3%
Optimization Iteration:   2901, Training Batch Accuracy: 100.0%
Optimization Iteration:   3001, Training Batch Accuracy:  98.4%
Optimization Iteration:   3101, Training Batch Accuracy:  93.8%
Optimization Iteration:   3201, Training Batch Accuracy:  98.4%
Optimization Iteration:   3301, Training Batch Accuracy:  96.9%
Optimization Iteration:   3401, Training Batch Accuracy: 100.0%
Optimization Iteration:   3501, Training Batch Accuracy:  95.3%
Optimization Iteration:   3601, Training Batch Accuracy:  98.4%
Optimization Iteration:   3701, Training Batch Accuracy:  98.4%
Optimization Iteration:   3801, Training Batch Accuracy: 100.0%
Optimization Iteration:   3901, Training Batch Accuracy:  95.3%
Optimization Iteration:   4001, Training Batch Accuracy: 100.0%
Optimization Iteration:   4101, Training Batch Accuracy: 100.0%
Optimization Iteration:   4201, Training Batch Accuracy: 100.0%
Optimization Iteration:   4301, Training Batch Accuracy:  98.4%
Optimization Iteration:   4401, Training Batch Accuracy:  98.4%
Optimization Iteration:   4501, Training Batch Accuracy: 100.0%
Optimization Iteration:   4601, Training Batch Accuracy: 100.0%
Optimization Iteration:   4701, Training Batch Accuracy:  98.4%
Optimization Iteration:   4801, Training Batch Accuracy: 100.0%
Optimization Iteration:   4901, Training Batch Accuracy: 100.0%
Optimization Iteration:   5001, Training Batch Accuracy:  98.4%
Optimization Iteration:   5101, Training Batch Accuracy:  96.9%
Optimization Iteration:   5201, Training Batch Accuracy: 100.0%
Optimization Iteration:   5301, Training Batch Accuracy: 100.0%
Optimization Iteration:   5401, Training Batch Accuracy:  98.4%
Optimization Iteration:   5501, Training Batch Accuracy:  98.4%
Optimization Iteration:   5601, Training Batch Accuracy:  98.4%
Optimization Iteration:   5701, Training Batch Accuracy: 100.0%
Optimization Iteration:   5801, Training Batch Accuracy: 100.0%
Optimization Iteration:   5901, Training Batch Accuracy:  98.4%
Optimization Iteration:   6001, Training Batch Accuracy: 100.0%
Optimization Iteration:   6101, Training Batch Accuracy: 100.0%
Optimization Iteration:   6201, Training Batch Accuracy: 100.0%
Optimization Iteration:   6301, Training Batch Accuracy:  98.4%
Optimization Iteration:   6401, Training Batch Accuracy: 100.0%
Optimization Iteration:   6501, Training Batch Accuracy:  98.4%
Optimization Iteration:   6601, Training Batch Accuracy: 100.0%
Optimization Iteration:   6701, Training Batch Accuracy: 100.0%
Optimization Iteration:   6801, Training Batch Accuracy:  98.4%
Optimization Iteration:   6901, Training Batch Accuracy: 100.0%
```

```
Optimization Iteration:   7001, Training Batch Accuracy:  98.4%
Optimization Iteration:   7101, Training Batch Accuracy: 100.0%
Optimization Iteration:   7201, Training Batch Accuracy: 100.0%
Optimization Iteration:   7301, Training Batch Accuracy:  98.4%
Optimization Iteration:   7401, Training Batch Accuracy:  96.9%
Optimization Iteration:   7501, Training Batch Accuracy: 100.0%
Optimization Iteration:   7601, Training Batch Accuracy:  98.4%
Optimization Iteration:   7701, Training Batch Accuracy:  98.4%
Optimization Iteration:   7801, Training Batch Accuracy: 100.0%
Optimization Iteration:   7901, Training Batch Accuracy:  98.4%
Optimization Iteration:   8001, Training Batch Accuracy:  98.4%
Optimization Iteration:   8101, Training Batch Accuracy:  96.9%
Optimization Iteration:   8201, Training Batch Accuracy: 100.0%
Optimization Iteration:   8301, Training Batch Accuracy:  98.4%
Optimization Iteration:   8401, Training Batch Accuracy: 100.0%
Optimization Iteration:   8501, Training Batch Accuracy: 100.0%
Optimization Iteration:   8601, Training Batch Accuracy:  98.4%
Optimization Iteration:   8701, Training Batch Accuracy: 100.0%
Optimization Iteration:   8801, Training Batch Accuracy: 100.0%
Optimization Iteration:   8901, Training Batch Accuracy: 100.0%
Optimization Iteration:   9001, Training Batch Accuracy: 100.0%
Optimization Iteration:   9101, Training Batch Accuracy:  98.4%
Optimization Iteration:   9201, Training Batch Accuracy:  98.4%
Optimization Iteration:   9301, Training Batch Accuracy: 100.0%
Optimization Iteration:   9401, Training Batch Accuracy: 100.0%
Optimization Iteration:   9501, Training Batch Accuracy: 100.0%
Optimization Iteration:   9601, Training Batch Accuracy:  98.4%
Optimization Iteration:   9701, Training Batch Accuracy:  95.3%
Optimization Iteration:   9801, Training Batch Accuracy:  96.9%
Optimization Iteration:   9901, Training Batch Accuracy: 100.0%
Time usage: 0:00:39

Neural network: 3
Optimization Iteration:      1, Training Batch Accuracy:   9.4%
Optimization Iteration:    101, Training Batch Accuracy:  89.1%
Optimization Iteration:    201, Training Batch Accuracy:  89.1%
Optimization Iteration:    301, Training Batch Accuracy:  90.6%
Optimization Iteration:    401, Training Batch Accuracy:  93.8%
Optimization Iteration:    501, Training Batch Accuracy:  93.8%
Optimization Iteration:    601, Training Batch Accuracy:  90.6%
Optimization Iteration:    701, Training Batch Accuracy:  96.9%
Optimization Iteration:    801, Training Batch Accuracy:  93.8%
Optimization Iteration:    901, Training Batch Accuracy:  96.9%
Optimization Iteration:   1001, Training Batch Accuracy:  98.4%
Optimization Iteration:   1101, Training Batch Accuracy: 100.0%
Optimization Iteration:   1201, Training Batch Accuracy: 100.0%
Optimization Iteration:   1301, Training Batch Accuracy:  98.4%
Optimization Iteration:   1401, Training Batch Accuracy:  96.9%
Optimization Iteration:   1501, Training Batch Accuracy:  96.9%
Optimization Iteration:   1601, Training Batch Accuracy:  98.4%
Optimization Iteration:   1701, Training Batch Accuracy:  92.2%
Optimization Iteration:   1801, Training Batch Accuracy:  96.9%
Optimization Iteration:   1901, Training Batch Accuracy:  98.4%
```

```
Optimization Iteration:   2001, Training Batch Accuracy:  93.8%
Optimization Iteration:   2101, Training Batch Accuracy:  98.4%
Optimization Iteration:   2201, Training Batch Accuracy: 100.0%
Optimization Iteration:   2301, Training Batch Accuracy: 100.0%
Optimization Iteration:   2401, Training Batch Accuracy: 100.0%
Optimization Iteration:   2501, Training Batch Accuracy: 100.0%
Optimization Iteration:   2601, Training Batch Accuracy: 100.0%
Optimization Iteration:   2701, Training Batch Accuracy:  96.9%
Optimization Iteration:   2801, Training Batch Accuracy: 100.0%
Optimization Iteration:   2901, Training Batch Accuracy:  95.3%
Optimization Iteration:   3001, Training Batch Accuracy: 100.0%
Optimization Iteration:   3101, Training Batch Accuracy: 100.0%
Optimization Iteration:   3201, Training Batch Accuracy: 100.0%
Optimization Iteration:   3301, Training Batch Accuracy:  98.4%
Optimization Iteration:   3401, Training Batch Accuracy:  98.4%
Optimization Iteration:   3501, Training Batch Accuracy:  98.4%
Optimization Iteration:   3601, Training Batch Accuracy: 100.0%
Optimization Iteration:   3701, Training Batch Accuracy: 100.0%
Optimization Iteration:   3801, Training Batch Accuracy:  95.3%
Optimization Iteration:   3901, Training Batch Accuracy:  98.4%
Optimization Iteration:   4001, Training Batch Accuracy: 100.0%
Optimization Iteration:   4101, Training Batch Accuracy:  98.4%
Optimization Iteration:   4201, Training Batch Accuracy: 100.0%
Optimization Iteration:   4301, Training Batch Accuracy:  95.3%
Optimization Iteration:   4401, Training Batch Accuracy: 100.0%
Optimization Iteration:   4501, Training Batch Accuracy: 100.0%
Optimization Iteration:   4601, Training Batch Accuracy: 100.0%
Optimization Iteration:   4701, Training Batch Accuracy:  95.3%
Optimization Iteration:   4801, Training Batch Accuracy:  98.4%
Optimization Iteration:   4901, Training Batch Accuracy:  98.4%
Optimization Iteration:   5001, Training Batch Accuracy:  98.4%
Optimization Iteration:   5101, Training Batch Accuracy:  98.4%
Optimization Iteration:   5201, Training Batch Accuracy: 100.0%
Optimization Iteration:   5301, Training Batch Accuracy: 100.0%
Optimization Iteration:   5401, Training Batch Accuracy: 100.0%
Optimization Iteration:   5501, Training Batch Accuracy: 100.0%
Optimization Iteration:   5601, Training Batch Accuracy: 100.0%
Optimization Iteration:   5701, Training Batch Accuracy: 100.0%
Optimization Iteration:   5801, Training Batch Accuracy: 100.0%
Optimization Iteration:   5901, Training Batch Accuracy: 100.0%
Optimization Iteration:   6001, Training Batch Accuracy: 100.0%
Optimization Iteration:   6101, Training Batch Accuracy: 100.0%
Optimization Iteration:   6201, Training Batch Accuracy:  95.3%
Optimization Iteration:   6301, Training Batch Accuracy: 100.0%
Optimization Iteration:   6401, Training Batch Accuracy:  98.4%
Optimization Iteration:   6501, Training Batch Accuracy: 100.0%
Optimization Iteration:   6601, Training Batch Accuracy:  98.4%
Optimization Iteration:   6701, Training Batch Accuracy: 100.0%
Optimization Iteration:   6801, Training Batch Accuracy: 100.0%
Optimization Iteration:   6901, Training Batch Accuracy: 100.0%
Optimization Iteration:   7001, Training Batch Accuracy: 100.0%
Optimization Iteration:   7101, Training Batch Accuracy: 100.0%
Optimization Iteration:   7201, Training Batch Accuracy: 100.0%
```

```
Optimization Iteration:   7301, Training Batch Accuracy: 100.0%
Optimization Iteration:   7401, Training Batch Accuracy: 100.0%
Optimization Iteration:   7501, Training Batch Accuracy: 100.0%
Optimization Iteration:   7601, Training Batch Accuracy: 100.0%
Optimization Iteration:   7701, Training Batch Accuracy: 100.0%
Optimization Iteration:   7801, Training Batch Accuracy:  98.4%
Optimization Iteration:   7901, Training Batch Accuracy: 100.0%
Optimization Iteration:   8001, Training Batch Accuracy:  98.4%
Optimization Iteration:   8101, Training Batch Accuracy: 100.0%
Optimization Iteration:   8201, Training Batch Accuracy: 100.0%
Optimization Iteration:   8301, Training Batch Accuracy: 100.0%
Optimization Iteration:   8401, Training Batch Accuracy: 100.0%
Optimization Iteration:   8501, Training Batch Accuracy: 100.0%
Optimization Iteration:   8601, Training Batch Accuracy: 100.0%
Optimization Iteration:   8701, Training Batch Accuracy: 100.0%
Optimization Iteration:   8801, Training Batch Accuracy: 100.0%
Optimization Iteration:   8901, Training Batch Accuracy: 100.0%
Optimization Iteration:   9001, Training Batch Accuracy: 100.0%
Optimization Iteration:   9101, Training Batch Accuracy:  98.4%
Optimization Iteration:   9201, Training Batch Accuracy: 100.0%
Optimization Iteration:   9301, Training Batch Accuracy: 100.0%
Optimization Iteration:   9401, Training Batch Accuracy: 100.0%
Optimization Iteration:   9501, Training Batch Accuracy:  96.9%
Optimization Iteration:   9601, Training Batch Accuracy:  98.4%
Optimization Iteration:   9701, Training Batch Accuracy:  98.4%
Optimization Iteration:   9801, Training Batch Accuracy:  98.4%
Optimization Iteration:   9901, Training Batch Accuracy: 100.0%
Time usage: 0:00:39

Neural network: 4
Optimization Iteration:      1, Training Batch Accuracy:   9.4%
Optimization Iteration:    101, Training Batch Accuracy:  82.8%
Optimization Iteration:    201, Training Batch Accuracy:  89.1%
Optimization Iteration:    301, Training Batch Accuracy:  89.1%
Optimization Iteration:    401, Training Batch Accuracy:  96.9%
Optimization Iteration:    501, Training Batch Accuracy:  96.9%
Optimization Iteration:    601, Training Batch Accuracy:  98.4%
Optimization Iteration:    701, Training Batch Accuracy:  96.9%
Optimization Iteration:    801, Training Batch Accuracy:  93.8%
Optimization Iteration:    901, Training Batch Accuracy:  96.9%
Optimization Iteration:   1001, Training Batch Accuracy:  98.4%
Optimization Iteration:   1101, Training Batch Accuracy:  96.9%
Optimization Iteration:   1201, Training Batch Accuracy:  93.8%
Optimization Iteration:   1301, Training Batch Accuracy:  96.9%
Optimization Iteration:   1401, Training Batch Accuracy:  98.4%
Optimization Iteration:   1501, Training Batch Accuracy:  95.3%
Optimization Iteration:   1601, Training Batch Accuracy:  96.9%
Optimization Iteration:   1701, Training Batch Accuracy:  98.4%
Optimization Iteration:   1801, Training Batch Accuracy:  93.8%
Optimization Iteration:   1901, Training Batch Accuracy:  96.9%
Optimization Iteration:   2001, Training Batch Accuracy: 100.0%
Optimization Iteration:   2101, Training Batch Accuracy:  95.3%
Optimization Iteration:   2201, Training Batch Accuracy:  96.9%
```

```
Optimization Iteration:   2301, Training Batch Accuracy:  96.9%
Optimization Iteration:   2401, Training Batch Accuracy:  93.8%
Optimization Iteration:   2501, Training Batch Accuracy:  98.4%
Optimization Iteration:   2601, Training Batch Accuracy:  96.9%
Optimization Iteration:   2701, Training Batch Accuracy:  98.4%
Optimization Iteration:   2801, Training Batch Accuracy:  98.4%
Optimization Iteration:   2901, Training Batch Accuracy:  98.4%
Optimization Iteration:   3001, Training Batch Accuracy:  96.9%
Optimization Iteration:   3101, Training Batch Accuracy:  98.4%
Optimization Iteration:   3201, Training Batch Accuracy:  98.4%
Optimization Iteration:   3301, Training Batch Accuracy: 100.0%
Optimization Iteration:   3401, Training Batch Accuracy:  98.4%
Optimization Iteration:   3501, Training Batch Accuracy:  96.9%
Optimization Iteration:   3601, Training Batch Accuracy:  98.4%
Optimization Iteration:   3701, Training Batch Accuracy:  98.4%
Optimization Iteration:   3801, Training Batch Accuracy:  98.4%
Optimization Iteration:   3901, Training Batch Accuracy: 100.0%
Optimization Iteration:   4001, Training Batch Accuracy:  98.4%
Optimization Iteration:   4101, Training Batch Accuracy: 100.0%
Optimization Iteration:   4201, Training Batch Accuracy: 100.0%
Optimization Iteration:   4301, Training Batch Accuracy: 100.0%
Optimization Iteration:   4401, Training Batch Accuracy:  98.4%
Optimization Iteration:   4501, Training Batch Accuracy:  98.4%
Optimization Iteration:   4601, Training Batch Accuracy:  98.4%
Optimization Iteration:   4701, Training Batch Accuracy: 100.0%
Optimization Iteration:   4801, Training Batch Accuracy: 100.0%
Optimization Iteration:   4901, Training Batch Accuracy:  96.9%
Optimization Iteration:   5001, Training Batch Accuracy: 100.0%
Optimization Iteration:   5101, Training Batch Accuracy:  98.4%
Optimization Iteration:   5201, Training Batch Accuracy: 100.0%
Optimization Iteration:   5301, Training Batch Accuracy: 100.0%
Optimization Iteration:   5401, Training Batch Accuracy:  98.4%
Optimization Iteration:   5501, Training Batch Accuracy: 100.0%
Optimization Iteration:   5601, Training Batch Accuracy:  98.4%
Optimization Iteration:   5701, Training Batch Accuracy:  95.3%
Optimization Iteration:   5801, Training Batch Accuracy: 100.0%
Optimization Iteration:   5901, Training Batch Accuracy: 100.0%
Optimization Iteration:   6001, Training Batch Accuracy:  98.4%
Optimization Iteration:   6101, Training Batch Accuracy: 100.0%
Optimization Iteration:   6201, Training Batch Accuracy:  96.9%
Optimization Iteration:   6301, Training Batch Accuracy:  98.4%
Optimization Iteration:   6401, Training Batch Accuracy: 100.0%
Optimization Iteration:   6501, Training Batch Accuracy: 100.0%
Optimization Iteration:   6601, Training Batch Accuracy: 100.0%
Optimization Iteration:   6701, Training Batch Accuracy: 100.0%
Optimization Iteration:   6801, Training Batch Accuracy:  96.9%
Optimization Iteration:   6901, Training Batch Accuracy:  96.9%
Optimization Iteration:   7001, Training Batch Accuracy: 100.0%
Optimization Iteration:   7101, Training Batch Accuracy: 100.0%
Optimization Iteration:   7201, Training Batch Accuracy:  96.9%
Optimization Iteration:   7301, Training Batch Accuracy: 100.0%
Optimization Iteration:   7401, Training Batch Accuracy: 100.0%
Optimization Iteration:   7501, Training Batch Accuracy: 100.0%
```

```
Optimization Iteration:   7601, Training Batch Accuracy: 100.0%
Optimization Iteration:   7701, Training Batch Accuracy: 100.0%
Optimization Iteration:   7801, Training Batch Accuracy: 100.0%
Optimization Iteration:   7901, Training Batch Accuracy:  93.8%
Optimization Iteration:   8001, Training Batch Accuracy: 100.0%
Optimization Iteration:   8101, Training Batch Accuracy:  98.4%
Optimization Iteration:   8201, Training Batch Accuracy: 100.0%
Optimization Iteration:   8301, Training Batch Accuracy: 100.0%
Optimization Iteration:   8401, Training Batch Accuracy: 100.0%
Optimization Iteration:   8501, Training Batch Accuracy:  98.4%
Optimization Iteration:   8601, Training Batch Accuracy: 100.0%
Optimization Iteration:   8701, Training Batch Accuracy:  98.4%
Optimization Iteration:   8801, Training Batch Accuracy: 100.0%
Optimization Iteration:   8901, Training Batch Accuracy:  98.4%
Optimization Iteration:   9001, Training Batch Accuracy: 100.0%
Optimization Iteration:   9101, Training Batch Accuracy: 100.0%
Optimization Iteration:   9201, Training Batch Accuracy: 100.0%
Optimization Iteration:   9301, Training Batch Accuracy: 100.0%
Optimization Iteration:   9401, Training Batch Accuracy: 100.0%
Optimization Iteration:   9501, Training Batch Accuracy: 100.0%
Optimization Iteration:   9601, Training Batch Accuracy: 100.0%
Optimization Iteration:   9701, Training Batch Accuracy: 100.0%
Optimization Iteration:   9801, Training Batch Accuracy: 100.0%
Optimization Iteration:   9901, Training Batch Accuracy:  98.4%
Time usage: 0:00:39
```

计算并且预测分类的帮助函数

这个函数计算了图像的预测标签，对每张图像来说，函数计算了一个长度为10的向量，向量显示了图像的类别。

计算分批完成，否则将占用太多内存。如果电脑死机了，你需要降低batch-size。

```
# Split the data-set in batches of this size to limit RAM usage.
batch_size = 256

def predict_labels(images):
    # Number of images.
    num_images = len(images)

    # Allocate an array for the predicted labels which
    # will be calculated in batches and filled into this array.
    pred_labels = np.zeros(shape=(num_images, num_classes),
                           dtype=np.float)

    # Now calculate the predicted labels for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing thi
s.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        # Create a feed-dict with the images between index i and
 j.
        feed_dict = {x: images[i:j, :]}

        # Calculate the predicted labels using TensorFlow.
        pred_labels[i:j] = session.run(y_pred, feed_dict=feed_di
ct)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    return pred_labels
```

计算一个布尔值向量，代表图像的预测类型是否正确。

```
def correct_prediction(images, labels, cls_true):
    # Calculate the predicted labels.
    pred_labels = predict_labels(images=images)

    # Calculate the predicted class-number for each image.
    cls_pred = np.argmax(pred_labels, axis=1)

    # Create a boolean array whether each image is correctly cla
ssified.
    correct = (cls_true == cls_pred)

    return correct
```

计算一个布尔数组，代表测试集中图像是否分类正确。

```
def test_correct():
    return correct_prediction(images = data.test.images,
                              labels = data.test.labels,
                              cls_true = data.test.cls)
```

计算一个布尔数组，代表验证集中图像是否分类正确。

```
def validation_correct():
    return correct_prediction(images = data.validation.images,
                              labels = data.validation.labels,
                              cls_true = data.validation.cls)
```

计算分类准确率的帮助函数

这个函数计算了给定布尔数组的分类准确率，布尔数组表示每张图像是否被正确分类。比如，
`cls_accuracy([True, True, False, False, False]) = 2/5 = 0.4` 。

```
def classification_accuracy(correct):
    # When averaging a boolean array, False means 0 and True mea
ns 1.
    # So we are calculating: number of True / len(correct) which
 is
    # the same as the classification accuracy.
    return correct.mean()
```

计算测试集的分类准确率。

```
def test_accuracy():
    # Get the array of booleans whether the classifications are
correct
    # for the test-set.
    correct = test_correct()

    # Calculate the classification accuracy and return it.
    return classification_accuracy(correct)
```

计算原始验证集上的分类准确率。

```
def validation_accuracy():
    # Get the array of booleans whether the classifications are
correct
    # for the validation-set.
    correct = validation_correct()

    # Calculate the classification accuracy and return it.
    return classification_accuracy(correct)
```

结果与分析

函数用来为ensemble中的所有神经网络计算预测标签。后面会将这些标签合并起来。

```
def ensemble_predictions():
    # Empty list of predicted labels for each of the neural netw
orks.
    pred_labels = []

    # Classification accuracy on the test-set for each network.
    test_accuracies = []

    # Classification accuracy on the validation-set for each net
work.
    val_accuracies = []

    # For each neural network in the ensemble.
    for i in range(num_networks):
        # Reload the variables into the TensorFlow graph.
        saver.restore(sess=session, save_path=get_save_path(i))

        # Calculate the classification accuracy on the test-set.
        test_acc = test_accuracy()

        # Append the classification accuracy to the list.
        test_accuracies.append(test_acc)

        # Calculate the classification accuracy on the validatio
n-set.
        val_acc = validation_accuracy()

        # Append the classification accuracy to the list.
        val_accuracies.append(val_acc)

        # Print status message.
        msg = "Network: {0}, Accuracy on Validation-Set: {1:.4f}
, Test-Set: {2:.4f}"
        print(msg.format(i, val_acc, test_acc))

        # Calculate the predicted labels for the images in the t
est-set.
        # This is already calculated in test_accuracy() above but

        # it is re-calculated here to keep the code a bit simple
r.
        pred = predict_labels(images=data.test.images)

        # Append the predicted labels to the list.
        pred_labels.append(pred)

    return np.array(pred_labels), \
           np.array(test_accuracies), \
           np.array(val_accuracies)
```

```
pred_labels, test_accuracies, val_accuracies = ensemble_predicti
ons()
```

```
Network: 0, Accuracy on Validation-Set: 0.9948, Test-Set: 0.9893
Network: 1, Accuracy on Validation-Set: 0.9936, Test-Set: 0.9880
Network: 2, Accuracy on Validation-Set: 0.9958, Test-Set: 0.9893
Network: 3, Accuracy on Validation-Set: 0.9938, Test-Set: 0.9889
Network: 4, Accuracy on Validation-Set: 0.9938, Test-Set: 0.9892
```

总结ensemble中的神经网络在测试集上的分类准确率。

```
print("Mean test-set accuracy: {0:.4f}".format(np.mean(test_accu
racies)))
print("Min test-set accuracy:  {0:.4f}".format(np.min(test_accur
acies)))
print("Max test-set accuracy:  {0:.4f}".format(np.max(test_accur
acies)))
```

```
Mean test-set accuracy: 0.9889
Min test-set accuracy:  0.9880
Max test-set accuracy:  0.9893
```

ensemble的预测标签是3维的数组，第一维是神经网络数量，第二维是图像数量，第三维是分类向量。

```
pred_labels.shape
```

```
(5, 10000, 10)
```

ensemble预测

有几种不同的方法来计算ensemble的预测标签。一种是计算每个神经网络的预测类别数字，然后选择得票最多的那个类别。但根据分类的类别数量，这种方法需要大量的神经网络。

这里用的方法是取ensemble中所有预测标签的平均。这个计算很简单，而且集成种不需要大量的神经网络。

```
ensemble_pred_labels = np.mean(pred_labels, axis=0)
ensemble_pred_labels.shape
```

```
(10000, 10)
```

取标签中最大数字的索引作为ensemble的预测类别数字，这通常用argmax来计算。

```
ensemble_cls_pred = np.argmax(ensemble_pred_labels, axis=1)
ensemble_cls_pred.shape
```

```
(10000,)
```

布尔数组表示测试集中的图像是否被神经网络的ensemble正确分类。

```
ensemble_correct = (ensemble_cls_pred == data.test.cls)
```

对布尔数组取反，因此我们可以用它来查找误分类的图像。

```
ensemble_incorrect = np.logical_not(ensemble_correct)
```

最佳的神经网络

现在我们找出在测试集上表现最佳的单个神经网络。

首先列出ensemble中所有神经网络在测试集上的分类准确率。

```
test_accuracies
```

```
array([ 0.9893,  0.988 ,  0.9893,  0.9889,  0.9892])
```

准确率最高的神经网络索引。

```
best_net = np.argmax(test_accuracies)
best_net
```

```
0
```

最佳神经网络在测试集上的分类准确率。

```
test_accuracies[best_net]
```

```
0.98929999999999996
```

最佳神经网络的预测标签。

```
best_net_pred_labels = pred_labels[best_net, :, :]
```

预测的类别数字。

```
best_net_cls_pred = np.argmax(best_net_pred_labels, axis=1)
```

最佳神经网络在测试集上是否正确分类图像的布尔数组。

```
best_net_correct = (best_net_cls_pred == data.test.cls)
```

图像是否被误分类的布尔数组。

```
best_net_incorrect = np.logical_not(best_net_correct)
```

ensemble与最佳网络的比较

测试集中被ensemble正确分类的图像数量。

```
np.sum(ensemble_correct)
```

```
9916
```

测试集中被最佳网络正确分类的图像数量。

```
np.sum(best_net_correct)
```

```
9893
```

布尔数组表示测试集中每张图像是否“被ensemble正确分类且被最佳网络误分类”。

```
ensemble_better = np.logical_and(best_net_incorrect,
                                 ensemble_correct)
```

测试集上ensemble比最佳网络表现更好的图像数量：

```
ensemble_better.sum()
```

```
39
```

布尔数组表示测试集中每张图像是否“被最佳网络正确分类且被ensemble误分类”。

```
best_net_better = np.logical_and(best_net_correct,
                                 ensemble_incorrect)
```

测试集上最佳网络比ensemble表现更好的图像数量：

```
best_net_better.sum()
```

```
16
```

绘制以及打印对比的帮助函数

函数用来绘制测试集中的图像，以及它们的真实类别与预测类别。

```
def plot_images_comparison(idx):
    plot_images(images=data.test.images[idx, :],
                cls_true=data.test.cls[idx],
                ensemble_cls_pred=ensemble_cls_pred[idx],
                best_cls_pred=best_net_cls_pred[idx])
```

打印预测标签的函数。

```
def print_labels(labels, idx, num=1):
    # Select the relevant labels based on idx.
    labels = labels[idx, :]

    # Select the first num labels.
    labels = labels[0:num, :]

    # Round numbers to 2 decimal points so they are easier to re
ad.
    labels_rounded = np.round(labels, 2)

    # Print the rounded labels.
    print(labels_rounded)
```

Function for printing the predicted labels for the ensemble of neural networks.

打印神经网络ensemble预测标签的函数。

```
def print_labels_ensemble(idx, **kwargs):
    print_labels(labels=ensemble_pred_labels, idx=idx, **kwargs)
```

打印单个网络预测标签的函数。

```
def print_labels_best_net(idx, **kwargs):
    print_labels(labels=best_net_pred_labels, idx=idx, **kwargs)
```

打印ensemble中所有神经网络预测标签的函数。只打印第一张图像的标签。

```
def print_labels_all_nets(idx):
    for i in range(num_networks):
        print_labels(labels=pred_labels[i, :, :], idx=idx, num=1
)
```

样本：ensemble比最佳网络好

绘制出那些被集成网络正确分类，且被最佳网络误分类的样本。

```
plot_images_comparison(idx=ensemble_better)
```

True: 3
Ensemble: 3
Best Net: 8

True: 4
Ensemble: 4
Best Net: 2

True: 5
Ensemble: 5
Best Net: 3

True: 5
Ensemble: 5
Best Net: 8

True: 3
Ensemble: 3
Best Net: 5

True: 6
Ensemble: 6
Best Net: 8

True: 9
Ensemble: 9
Best Net: 5

True: 7
Ensemble: 7
Best Net: 2

True: 2
Ensemble: 2
Best Net: 3

ensemble对第一张图像（左上）的预测标签：

```
print_labels_ensemble(idx=ensemble_better, num=1)
```

```
[[ 0.    0.    0.    0.76  0.    0.    0.    0.    0.23  0.  ]]
```

最佳网络对第一张图像的预测标签：

```
print_labels_best_net(idx=ensemble_better, num=1)
```

```
[[ 0.    0.    0.    0.21  0.    0.    0.    0.    0.79  0.  ]]
```

ensemble中所有网络对第一张图像的预测标签：

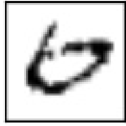
```
print_labels_all_nets(idx=ensemble_better)
```

```
[[ 0.    0.    0.    0.21  0.    0.    0.    0.    0.79  0.  ]]
[[ 0.    0.    0.    0.96  0.    0.01  0.    0.    0.03  0.  ]]
[[ 0.    0.    0.    0.99  0.    0.    0.    0.    0.01  0.  ]]
[[ 0.    0.    0.    0.88  0.    0.    0.    0.    0.12  0.  ]]
[[ 0.    0.    0.    0.76  0.    0.01  0.    0.    0.22  0.  ]]
```

样本：最佳网络比ensemble好

现在绘制那些被ensemble误分类，但被最佳网络正确分类的样本。

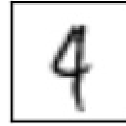
```
plot_images_comparison(idx=best_net_better)
```



True: 6
Ensemble: 0
Best Net: 6



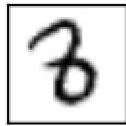
True: 7
Ensemble: 3
Best Net: 7



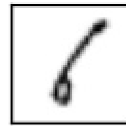
True: 4
Ensemble: 9
Best Net: 4



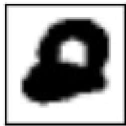
True: 9
Ensemble: 3
Best Net: 9



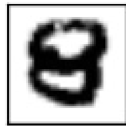
True: 8
Ensemble: 3
Best Net: 8



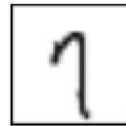
True: 6
Ensemble: 1
Best Net: 6



True: 2
Ensemble: 0
Best Net: 2



True: 8
Ensemble: 9
Best Net: 8



True: 7
Ensemble: 9
Best Net: 7

ensemble对第一张图像（左上）的预测标签：

```
print_labels_ensemble(idx=best_net_better, num=1)
```

```
[[ 0.5   0.    0.    0.    0.    0.05  0.45  0.    0.    0.  ]]
```

最佳网络对第一张图像的预测标签：

```
print_labels_best_net(idx=best_net_better, num=1)
```

```
[[ 0.3   0.    0.    0.    0.    0.15  0.56  0.    0.    0.  ]]
```

ensemble中所有网络对第一张图像的预测标签：

```
print_labels_all_nets(idx=best_net_better)
```

```
[[ 0.3   0.    0.    0.    0.    0.15  0.56  0.    0.    0.  ]]
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
[[ 0.19  0.    0.    0.    0.    0.    0.81  0.    0.    0.  ]]
[[ 0.15  0.    0.    0.    0.    0.12  0.72  0.    0.    0.  ]]
[[ 0.85  0.    0.    0.    0.    0.    0.14  0.    0.    0.  ]]
```

关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# session.close()
```

总结

这篇教程创建了5个神经网络的集成（ensemble），用来识别MINIST数据集中的手写数字。ensemble取5个单独神经网络的平均值。最终稍微提高了在测试集上的分类准确率，相比单个最佳网络98.9%的准确率，ensemble是99.1%。

然而，ensemble的表现并不是一直都比单个网络好，有些单个网络正确分类的图像却被ensemble误分类。这表明神经网络ensemble的作用有点随机，可能无法提供一个提升性能的可靠方式（和单独神经网络性能相比）。

这里使用的集成学习的形式叫bagging (或 Bootstrap Aggregating)，它常用来避免过拟合，但对（本文中的）这个特定的神经网络和数据集来说不是必要的。在其他情况下集成学习可能仍然有效。

技术说明

本文在实现集成学习时用了TensorFlow中 `Saver()` 对象来保存和恢复神经网络中的变量。但这个功能其实是为其他目的设计的，使用在有多种类型神经网络的集成学习中，或者想同时载入多个神经网络时就有点笨拙了。有一个叫 sk-flow 的 TensorFlow添加包有更简单的方法，但到2016年八月为止，它仍然处于开发的前期阶段。

练习

下面是一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 改变程序的几个不同地方，看看它如何影响性能：
    - 在集成中使用更多神经网络。
    - 改变训练集的大小。
    - 改变优化迭代的次数，试着增加或减少。
- 向朋友解释程序如何工作。
- 你认为集成学习值得更多的研究吗，或者宁可专注于提升单个神经网络的性能？

License (MIT)

TensorFlow 教程 #06

CIFAR-10

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

简介

这篇教程介绍了如何创建一个在CIRAR-10数据集上进行图像分类的卷积神经网络。同时也说明了在训练和测试时如何使用不同的网络。

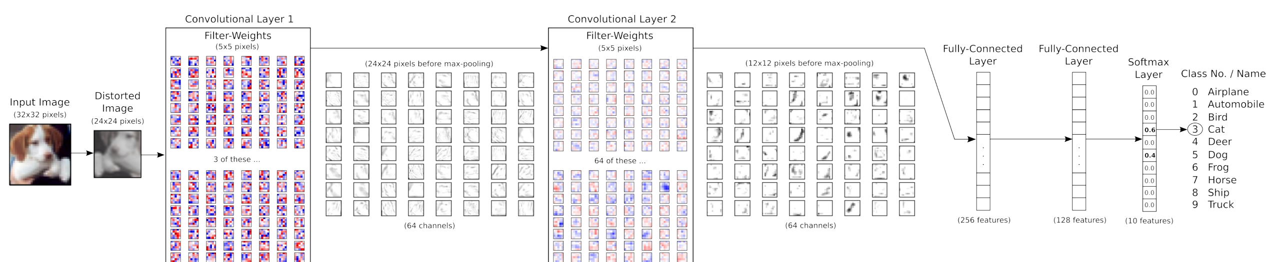
本文基于上一篇教程，你需要了解基本的TensorFlow和附加包Pretty Tensor。其中大量代码和文字与之前教程相似，如果你已经看过可以快速地浏览本文。

流程图

下面的图表直接显示了之后实现的卷积神经网络中数据的传递。首先有一个扭曲（distorts）输入图像的预处理层，用来人为地扩大训练集。接着有两个卷积层，两个全连接层和一个softmax分类层。在后面会有更大的图示来显示权重和卷积层的输出，教程 #02 有卷积如何工作的更多细节。

在这种情况下图像是误分类的。图像上有一只狗，但神经网络不确定它是狗还是猫，认为更有可能是猫。

```
from IPython.display import Image
Image('images/06_network_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
import time
from datetime import timedelta
import math
import os

# Use PrettyTensor to simplify Neural Network construction.
import prettytensor as pt
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.12.0-rc0'
```

PrettyTensor 版本:

```
pt.__version__
```

```
'0.7.1'
```

载入数据

```
import cifar10
```

设置电脑上保存数据集的路径。

```
# cifar10.data_path = "data/CIFAR-10/"
```

CIFAR-10数据集大概有163MB，如果给定路径没有找到文件的话，将会自动下载。

```
cifar10.maybe_download_and_extract()
```

```
Data has apparently already been downloaded and unpacked.
```

载入分类名称。

```
class_names = cifar10.load_class_names()
class_names
```

```
Loading data: data/CIFAR-10/cifar-10-batches-py/batches.meta




['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']
```

载入训练集。这个函数返回图像、整形分类号码、以及用One-Hot编码的分类号数组，称为标签。

```
images_train, cls_train, labels_train = cifar10.load_training_da
ta()
```

```
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_1
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_2
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_3
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_4
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_5
```

载入测试集。

```
images_test, cls_test, labels_test = cifar10.load_test_data()
```

```
Loading data: data/CIFAR-10/cifar-10-batches-py/test_batch
```

现在已经载入了CIFAR-10数据集，它包含60,000张图像以及相关的标签（图像的分类）。数据集被分为两个独立的子集，即训练集和测试集。

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(images_train)))
print("- Test-set:\t\t{}".format(len(images_test)))
```

```
Size of:
- Training-set:		50000
- Test-set:		10000
```

数据维度

下面的代码中多次用到数据维度。cirfa10模块中已经定义好了这些，因此我们只需要import进来。

```
from cifar10 import img_size, num_channels, num_classes
```

图像是32 x 32像素的，但我们将图像裁剪至24 x 24像素。

```
img_size_cropped = 24
```

用来绘制图片的帮助函数

这个函数用来在3x3的栅格中画9张图像，然后在每张图像下面写出真实类别和预测类别。

```
def plot_images(images, cls_true, cls_pred=None, smooth=True):

    assert len(images) == len(cls_true) == 9

    # Create figure with sub-plots.
    fig, axes = plt.subplots(3, 3)

    # Adjust vertical spacing if we need to print ensemble and b
est-net.
    if cls_pred is None:
        hspace = 0.3
    else:
        hspace = 0.6
    fig.subplots_adjust(hspace=hspace, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Interpolation type.
        if smooth:
            interpolation = 'spline16'
        else:
            interpolation = 'nearest'

        # Plot image.
        ax.imshow(images[i, :, :, :],
                  interpolation=interpolation)

        # Name of the true class.
        cls_true_name = class_names[cls_true[i]]

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true_name)
        else:
            # Name of the predicted class.
            cls_pred_name = class_names[cls_pred[i]]

            xlabel = "True: {0}\nPred: {1}".format(cls_true_name
, cls_pred_name)

        # Show the classes as the label on the x-axis.
        ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

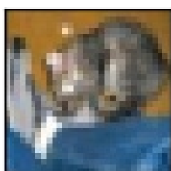
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

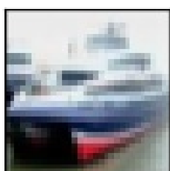
绘制几张图像来看看数据是否正确

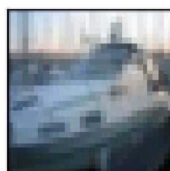
```
# Get the first images from the test-set.
images = images_test[0:9]

# Get the true classes for those images.
cls_true = cls_test[0:9]

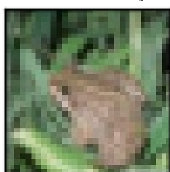
# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true, smooth=False)
```

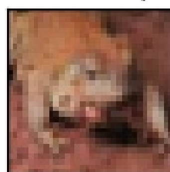

True: cat


True: ship

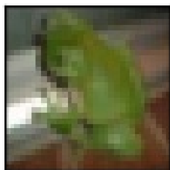

True: ship

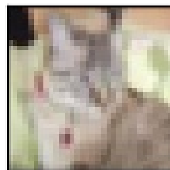

True: airplane


True: frog


True: frog

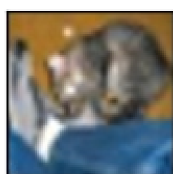

True: automobile

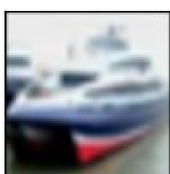

True: frog

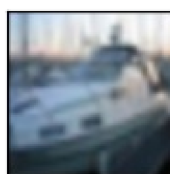

True: cat

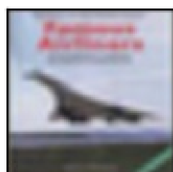
上面像素化的图像是神经网络的输入。如果我们对图像进行平滑处理，可能更易于人眼识别。

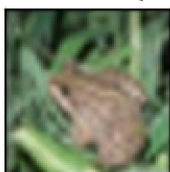
```
plot_images(images=images, cls_true=cls_true, smooth=True)
```

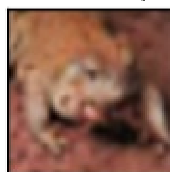

True: cat

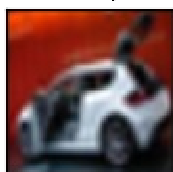

True: ship

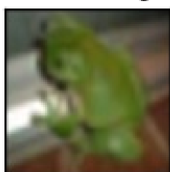

True: ship

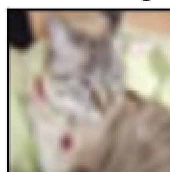

True: airplane


True: frog


True: frog


True: automobile


True: frog


True: cat

TensorFlow图

TensorFlow的全部目的就是使用一个称之为计算图（computational graph）的东西，它会比直接在Python中进行相同计算量要高效得多。TensorFlow比Numpy更高效，因为TensorFlow了解整个需要运行的计算图，然而Numpy只知道某个时间点上唯一的数学运算。

TensorFlow也能够自动地计算需要优化的变量的梯度，使得模型有更好的表现。这是由于图是简单数学表达式的结合，因此整个图的梯度可以用链式法则推导出来。

TensorFlow还能利用多核CPU和GPU，Google也为TensorFlow制造了称为TPUs（Tensor Processing Units）的特殊芯片，它比GPU更快。

一个TensorFlow图由下面几个部分组成，后面会详细描述：

- 占位符变量（Placeholder）用来改变图的输入。
- 模型变量（Model）将会被优化，使得模型表现得更好。
- 模型本质上就是一些数学函数，它根据Placeholder和模型的输入变量来计算一些输出。
- 一个cost度量用来指导变量的优化。
- 一个优化策略会更新模型的变量。

另外，TensorFlow图也包含了一些调试状态，比如用TensorBoard打印log数据，本教程不涉及这些。

占位符（Placeholder）变量

Placeholder是作为图的输入，我们每次运行图的时候都可能改变它们。将这个过程称为feeding placeholder变量，后面将会描述这个。

首先我们为输入图像定义placeholder变量。这让我们可以改变输入到TensorFlow图中的图像。这也是一个张量（tensor），代表一个多维向量或矩阵。数据类型设置为float32，形状设为 `[None, img_size, img_size, num_channels]` 代表tensor可能保存着任意数量的图像，每张图像宽高都为 `img_size` ，有 `num_channels` 个颜色通道。

```
x = tf.placeholder(tf.float32, shape=[None, img_size, img_size,
num_channels], name='x')
```

接下来我们为输入变量 `x` 中的图像所对应的真实标签定义placeholder变量。变量的形状是 `[None, num_classes]` ，这代表着它保存了任意数量的标签，每个标签是长度为 `num_classes` 的向量，本例中长度为10。

```
y_true = tf.placeholder(tf.float32, shape=[None, num_classes], n
ame='y_true')
```

我们也可以为class-number提供一个placeholder，但这里用argmax来计算它。这里只是TensorFlow中的一些操作，没有执行什么运算。

```
y_true_cls = tf.argmax(y_true, dimension=1)
```

预处理的帮助函数

下面的帮助函数创建了用来预处理输入图像的TensorFlow计算图。这里并未执行计算，函数只是给TensorFlow计算图添加了节点。

神经网络在训练和测试阶段的预处理方法不同：

- 对于训练来说，输入图像是随机裁剪、水平翻转的，并且用随机值来调整色调、对比度和饱和度。这样就创建了原始输入图像的随机变体，人为地扩充了训练集。后面会显示一些扭曲过的图像样本。
- 对于测试，输入图像根据中心裁剪，其他不作调整。

```
def pre_process_image(image, training):
    # This function takes a single image as input,
    # and a boolean whether to build the training or testing gra
ph.

    if training:
        # For training, add the following to the TensorFlow grap
h.

        # Randomly crop the input image.
        image = tf.random_crop(image, size=[img_size_cropped, im
g_size_cropped, num_channels])

        # Randomly flip the image horizontally.
        image = tf.image.random_flip_left_right(image)

        # Randomly adjust hue, contrast and saturation.
        image = tf.image.random_hue(image, max_delta=0.05)
        image = tf.image.random_contrast(image, lower=0.3, upper=
1.0)
        image = tf.image.random_brightness(image, max_delta=0.2)
        image = tf.image.random_saturation(image, lower=0.0, upp
er=2.0)

        # Some of these functions may overflow and result in pix
el
        # values beyond the [0, 1] range. It is unclear from the
        # documentation of TensorFlow 0.10.0rc0 whether this is
        # intended. A simple solution is to limit the range.

        # Limit the image pixels between [0, 1] in case of overf
low.
        image = tf.minimum(image, 1.0)
        image = tf.maximum(image, 0.0)
    else:
        # For training, add the following to the TensorFlow grap
h.

        # Crop the input image around the centre so it is the sa
me
        # size as images that are randomly cropped during traini
ng.
        image = tf.image.resize_image_with_crop_or_pad(image,
                                                       target_he
ight=img_size_cropped,
                                                       target_wi
dth=img_size_cropped)

    return image
```

下面函数中，输入batch中每张图像都调用以上函数。

```
def pre_process(images, training):
    # Use TensorFlow to loop over all the input images and call
    # the function above which takes a single image as input.
    images = tf.map_fn(lambda image: pre_process_image(image, tr
aining), images)

    return images
```

为了绘制扭曲过的图像，我们为TensorFlow创建预处理graph，后面将会运行它。

```
distorted_images = pre_process(images=x, training=True)
```

创建主要处理程序的帮助函数

下面的帮助函数创建了卷积神经网络的主要部分。这里使用之前教程描述过的Pretty Tensor。

```
def main_network(images, training):
    # Wrap the input images as a Pretty Tensor object.
    x_pretty = pt.wrap(images)

    # Pretty Tensor uses special numbers to distinguish between
    # the training and testing phases.
    if training:
        phase = pt.Phase.train
    else:
        phase = pt.Phase.infer

    # Create the convolutional neural network using Pretty Tenso
r.
    # It is very similar to the previous tutorials, except
    # the use of so-called batch-normalization in the first laye
r.
    with pt.defaults_scope(activation_fn=tf.nn.relu, phase=phase
):
        y_pred, loss = x_pretty.\
            conv2d(kernel=5, depth=64, name='layer_conv1', batch
_normalize=True).\
            max_pool(kernel=2, stride=2).\
            conv2d(kernel=5, depth=64, name='layer_conv2').\
            max_pool(kernel=2, stride=2).\
            flatten().\
            fully_connected(size=256, name='layer_fc1').\
            fully_connected(size=128, name='layer_fc2').\
            softmax_classifier(num_classes=num_classes, labels=y
_true)

    return y_pred, loss
```

创建神经网络的帮助函数

下面的帮助函数创建了整个神经网络，包含上面定义的预处理以及主要处理模块。

注意，神经网络被编码到'network'变量作用域中。因为我们实际上在TensorFlow图中创建了两个神经网络。像这样指定一个变量作用域，可以在两个神经网络中复用变量，因此训练网络优化过的变量可以在测试网络中复用。

```
def create_network(training):
    # Wrap the neural network in the scope named 'network'.
    # Create new variables during training, and re-use during te
sting.
    with tf.variable_scope('network', reuse=not training):
        # Just rename the input placeholder variable for conveni
ence.
        images = x

        # Create TensorFlow graph for pre-processing.
        images = pre_process(images=images, training=training)

        # Create TensorFlow graph for the main processing.
        y_pred, loss = main_network(images=images, training=trai
ning)

    return y_pred, loss
```

为训练阶段创建神经网络

首先创建一个保存当前优化迭代次数的TensorFlow变量。在之前的教程中，是使用一个Python变量，但本教程中，我们想用checkpoints中的其他TensorFlow变量来保存。

`trainable=False` 表示TensorFlow不会优化此变量。

```
global_step = tf.Variable(initial_value=0,
                          name='global_step', trainable=False)
```

创建训练用的神经网络。函数 `create_network()` 返回 `y_pred` 和 `loss` ，但在训练时我们只需用到 `loss` 函数。

```
_, loss = create_network(training=True)
```

创建最小化 `loss` 函数的优化器。同时将 `global_step` 传给优化器，这样每次迭代它都减一。

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(
loss, global_step=global_step)
```

创建测试阶段的神经网络

现在创建测试阶段的神经网络。 同样的， `create_network()` 返回输入图像的预测标签 `y_pred` ，优化过程也用到 `loss` 函数。测试时我们只需要 `y_pred` 。

```
y_pred, _ = create_network(training=False)
```

然后我们计算预测类别号的整形数字。网络的输出 `y_pred` 是一个10个元素的数组。类别号是数组中最大元素的索引。

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

然后创建一个布尔向量，用来告诉我们每张图片的真实类别是否与预测类别相同。

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

上面的计算先将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确率。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
))
```

Saver

为了保存神经网络的变量（这样不必再次训练网络就能重载），我们创建一个称为Saver-object的对象，它用来保存及恢复TensorFlow图的所有变量。在这里并未保存什么东西，（保存操作）在后面的 `optimize()` 函数中完成。

```
saver = tf.train.Saver()
```

获取权重

下面，我们要绘制神经网络的权重。当使用Pretty Tensor来创建网络时，层的所有变量都是由Pretty Tensoe间接创建的。因此我们要从TensorFlow中获取变量。

我们用 `layer_conv1` 和 `layer_conv2` 代表两个卷积层。这也叫变量作用域（不要与上面描述的 `defaults_scope` 混淆了）。PrettyTensor会自动给它为每个层创建的变量命名，因此我们可以通过层的作用域名称和变量名来取得某一层的权重。

函数实现有点笨拙，因为我们不得不用TensorFlow函数 `get_variable()` ，它是设计给其他用途的，创建新的变量或重用现有变量。创建下面的帮助函数很简单。

```
def get_weights_variable(layer_name):
    # Retrieve an existing variable named 'weights' in the scope
    # with the given layer_name.
    # This is awkward because the TensorFlow function was
    # really intended for another purpose.

    with tf.variable_scope("network/" + layer_name, reuse=True):
        variable = tf.get_variable('weights')

    return variable
```

借助这个帮助函数我们可以获取变量。这些是TensorFlow的objects。你需要类似的操作来获取变量的内容： `contents = session.run(weights_conv1)` ，下面会提到这个。

```
weights_conv1 = get_weights_variable(layer_name='layer_conv1')
weights_conv2 = get_weights_variable(layer_name='layer_conv2')
```

获取layer的输出

同样的，我们还需要获取卷积层的输出。这个函数与上面获取权重的函数有所不同。这里我们找回卷积层输出的最后一个张量。

```
def get_layer_output(layer_name):
    # The name of the last operation of the convolutional layer.
    # This assumes you are using Relu as the activation-function.

    tensor_name = "network/" + layer_name + "/Relu:0"

    # Get the tensor with this name.
    tensor = tf.get_default_graph().get_tensor_by_name(tensor_na
me)

    return tensor
```

取得卷积层的输出以便之后绘制。

```
output_conv1 = get_layer_output(layer_name='layer_conv1')
output_conv2 = get_layer_output(layer_name='layer_conv2')
```

运行TensorFlow

创建TensorFlow会话（session）

一旦创建了TensorFlow图，我们需要创建一个TensorFlow会话，用来运行图。

```
session = tf.Session()
```

初始化或恢复变量

训练神经网络会花上很长时间，特别是当你没有GPU的时候。因此我们在训练时保存checkpoints，这样就能在其他时间继续训练（比如晚上），以后也可以不用训练神经网络就用这些来分析结果。

如果你想重新训练神经网络，就需要先删掉这些checkpoints。

这是用来保存checkpoints的文件夹。

```
save_dir = 'checkpoints/'
```

如果文件夹不存在则创建。

```
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
```

这是checkpoints的基本文件名，TensorFlow会在后面添加迭代次数等。

```
save_path = os.path.join(save_dir, 'cifar10_cnn')
```

试着载入最新的checkpoint。如果checkpoint不存在或改变了TensorFlow图的话，可能会失败并抛出异常。

```
try:
    print("Trying to restore last checkpoint ...")

    # Use TensorFlow to find the latest checkpoint - if any.
    last_chk_path = tf.train.latest_checkpoint(checkpoint_dir=sa
ve_dir)

    # Try and load the data in the checkpoint.
    saver.restore(session, save_path=last_chk_path)

    # If we get to this point, the checkpoint was successfully l
oaded.
    print("Restored checkpoint from:", last_chk_path)
except:
    # If the above failed for some reason, simply
    # initialize all the variables for the TensorFlow graph.
    print("Failed to restore checkpoint. Initializing variables
instead.")
    session.run(tf.global_variables_initializer())
```

```
Trying to restore last checkpoint ...
Restored checkpoint from: checkpoints/cifar10_cnn-150000
```

创建随机训练batch的帮助函数

在训练集中有50,000张图。用这些图像计算模型的梯度会花很多时间。因此，在优化器的每次迭代里只用到了一小部分的图像。

如果内存耗尽导致电脑死机或变得很慢，你应该试着减少这些数量，但同时可能还需要更优化的迭代。

```
train_batch_size = 64
```

函数从训练集中挑选一个随机的training-batch。

```
def random_batch():
    # Number of images in the training-set.
    num_images = len(images_train)

    # Create a random index.
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)

    # Use the random index to select random images and labels.
    x_batch = images_train[idx, :, :, :]
    y_batch = labels_train[idx, :]

    return x_batch, y_batch
```

执行优化迭代的帮助函数

函数用来执行一定数量的优化迭代，以此来逐渐改善网络层的变量。在每次迭代中，会从训练集中选择新的一批数据，然后TensorFlow在这些训练样本上执行优化。每100次迭代会打印出进度。每1000次迭代后会保存一个checkpoint，最后一次迭代完毕也会保存。

```
def optimize(num_iterations):
    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(num_iterations):
        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = random_batch()

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimiz
er.
        # We also want to retrieve the global_step counter.
        i_global, _ = session.run([global_step, optimizer],
                                  feed_dict=feed_dict_train)

        # Print status to screen every 100 iterations (and last).

        if (i_global % 100 == 0) or (i == num_iterations - 1):
            # Calculate the accuracy on the training-batch.
            batch_acc = session.run(accuracy,
```

```
                                                  feed_dict=feed_dict_train)

                # Print status.
                msg = "Global Step: {0:>6}, Training Batch Accuracy:
 {1:>6.1%}"
                print(msg.format(i_global, batch_acc))

            # Save a checkpoint to disk every 1000 iterations (and l
ast).
            if (i_global % 1000 == 0) or (i == num_iterations - 1):
                # Save all variables of the TensorFlow graph to a
                # checkpoint. Append the global_step counter
                # to the filename so we save the last several checkp
oints.
                saver.save(session,
                           save_path=save_path,
                           global_step=global_step)

                print("Saved checkpoint.")

        # Ending time.
        end_time = time.time()

        # Difference between start and end-times.
        time_dif = end_time - start_time

        # Print the time-usage.
        print("Time usage: " + str(timedelta(seconds=int(round(time_
dif)))))
```

用来绘制错误样本的帮助函数

函数用来绘制测试集中被误分类的样本。

```
def plot_example_errors(cls_pred, correct):
    # This function is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # correct is a boolean array whether the predicted class
    # is equal to the true class for each image in the test-set.

    # Negate the boolean array.
    incorrect = (correct == False)

    # Get the images from the test-set that have been
    # incorrectly classified.
    images = images_test[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = cls_test[incorrect]

    # Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])
```

绘制混淆（confusion）矩阵的帮助函数

```
def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_test,  # True class for tes
t-set.
                          y_pred=cls_pred)  # Predicted class.

    # Print the confusion matrix as text.
    for i in range(num_classes):
        # Append the class-name to each line.
        class_name = "({}) {}".format(i, class_names[i])
        print(cm[i, :], class_name)

    # Print the class-numbers for easy reference.
    class_numbers = [" ({0})".format(i) for i in range(num_class
es)]
    print("".join(class_numbers))
```

计算分类的帮助函数

这个函数用来计算图像的预测类别，同时返回一个代表每张图像分类是否正确的布尔数组。

由于计算可能会耗费太多内存，就分批处理。如果你的电脑死机了，试着降低batch-size。

```
# Split the data-set in batches of this size to limit RAM usage.
batch_size = 256

def predict_cls(images, labels, cls_true):
    # Number of images.
    num_images = len(images)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_images, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing thi
s.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        # Create a feed-dict with the images and labels
        # between index i and j.
        feed_dict = {x: images[i:j, :],
                     y_true: labels[i:j, :]}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_d
ict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Create a boolean array whether each image is correctly cla
ssified.
    correct = (cls_true == cls_pred)

    return correct, cls_pred
```

Calculate the predicted class for the test-set.

```
def predict_cls_test():
    return predict_cls(images = images_test,
                       labels = labels_test,
                       cls_true = cls_test)
```

计算分类准确率的帮助函数

这个函数计算了给定布尔数组的分类准确率，布尔数组表示每张图像是否被正确分类。比如，
`cls_accuracy([True, True, False, False, False]) = 2/5 = 0.4`。这个函数也返回了正确分类的数量。

```
def classification_accuracy(correct):
    # When averaging a boolean array, False means 0 and True means 1.
    # So we are calculating: number of True / len(correct) which is
    # the same as the classification accuracy.

    # Return the classification accuracy
    # and the number of correct classifications.
    return correct.mean(), correct.sum()
```

展示性能的帮助函数

函数用来打印测试集上的分类准确率。

为测试集上的所有图片计算分类会花费一段时间，因此我们直接从这个函数里调用上面的函数，这样就不用每个函数都重新计算分类。

```
def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # For all the images in the test-set,
    # calculate the predicted classes and whether they are corre
ct.
    correct, cls_pred = predict_cls_test()

    # Classification accuracy and the number of correct classifi
cations.
    acc, num_correct = classification_accuracy(correct)

    # Number of images being classified.
    num_images = len(correct)

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, num_correct, num_images))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

绘制卷积权重的帮助函数

```
def plot_conv_weights(weights, input_channel=0):
    # Assume weights are TensorFlow ops for 4-dim variables
    # e.g. weights_conv1 or weights_conv2.

    # Retrieve the values of the weight-variables from TensorFlo
w.
    # A feed-dict is not necessary because nothing is calculated.

    w = session.run(weights)

    # Print statistics for the weights.
    print("Min:  {0:.5f}, Max:   {1:.5f}".format(w.min(), w.max(
)))
    print("Mean: {0:.5f}, Stdev: {1:.5f}".format(w.mean(), w.std
()))

    # Get the lowest and highest values for the weights.
    # This is used to correct the colour intensity across
    # the images so they can be compared with each other.
```

```
    w_min = np.min(w)
    w_max = np.max(w)
    abs_max = max(abs(w_min), abs(w_max))

    # Number of filters used in the conv. layer.
    num_filters = w.shape[3]

    # Number of grids to plot.
    # Rounded-up, square-root of the number of filters.
    num_grids = math.ceil(math.sqrt(num_filters))

    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(num_grids, num_grids)

    # Plot all the filter-weights.
    for i, ax in enumerate(axes.flat):
        # Only plot the valid filter-weights.
        if i<num_filters:
            # Get the weights for the i'th filter of the input c
hannel.
            # The format of this 4-dim tensor is determined by t
he
            # TensorFlow API. See Tutorial #02 for more details.
            img = w[:, :, input_channel, i]

            # Plot image.
            ax.imshow(img, vmin=-abs_max, vmax=abs_max,
                      interpolation='nearest', cmap='seismic')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

绘制卷积层输出的帮助函数

```
def plot_layer_output(layer_output, image):
    # Assume layer_output is a 4-dim tensor
    # e.g. output_conv1 or output_conv2.

    # Create a feed-dict which holds the single input image.
    # Note that TensorFlow needs a list of images,
    # so we just create a list with this one image.
    feed_dict = {x: [image]}

    # Retrieve the output of the layer after inputting this imag
e.
    values = session.run(layer_output, feed_dict=feed_dict)

    # Get the lowest and highest values.
    # This is used to correct the colour intensity across
    # the images so they can be compared with each other.
    values_min = np.min(values)
    values_max = np.max(values)

    # Number of image channels output by the conv. layer.
    num_images = values.shape[3]

    # Number of grid-cells to plot.
    # Rounded-up, square-root of the number of filters.
    num_grids = math.ceil(math.sqrt(num_images))

    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(num_grids, num_grids)

    # Plot all the filter-weights.
    for i, ax in enumerate(axes.flat):
        # Only plot the valid image-channels.
        if i<num_images:
            # Get the images for the i'th output channel.
            img = values[0, :, :, i]

            # Plot image.
            ax.imshow(img, vmin=values_min, vmax=values_max,
                      interpolation='nearest', cmap='binary')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

输入图像变体的样本

为了人为地增加训练用的图像数量，神经网络预处理获取输入图像的随机变体。这让神经网络在识别和分类图像时更加灵活。

这是用来绘制输入图像变体的帮助函数。

```
def plot_distorted_image(image, cls_true):
    # Repeat the input image 9 times.
    image_duplicates = np.repeat(image[np.newaxis, :, :, :], 9, 
axis=0)

    # Create a feed-dict for TensorFlow.
    feed_dict = {x: image_duplicates}

    # Calculate only the pre-processing of the TensorFlow graph
    # which distorts the images in the feed-dict.
    result = session.run(distorted_images, feed_dict=feed_dict)

    # Plot the images.
    plot_images(images=result, cls_true=np.repeat(cls_true, 9))
```

帮助函数获取测试集图像以及它的分类号。

```
def get_test_image(i):
    return images_test[i, :, :, :], cls_test[i]
```

从测试集中取一张图像以及它的真实类别。

```
img, cls = get_test_image(16)
```

画出图像的9张随机变体。如果你重新运行代码，可能会得到不太一样的结果。

```
plot_distorted_image(img, cls)
```



执行优化

我的笔记本电脑是4核的，每个2GHz。电脑带有一个GPU，但对TensorFlow来说不太快，因此只用了CPU。在电脑上迭代10,000次大概花了1个小时。本教程中我执行了150,000次优化迭代，共花了15个小时。我让它在夜里以及白天的几个时间段运行。

由于我们在优化过程中保存了checkpoints，重新运行代码时会载入最后的那个checkpoint，所以可以先停止，等晚点再继续执行优化。

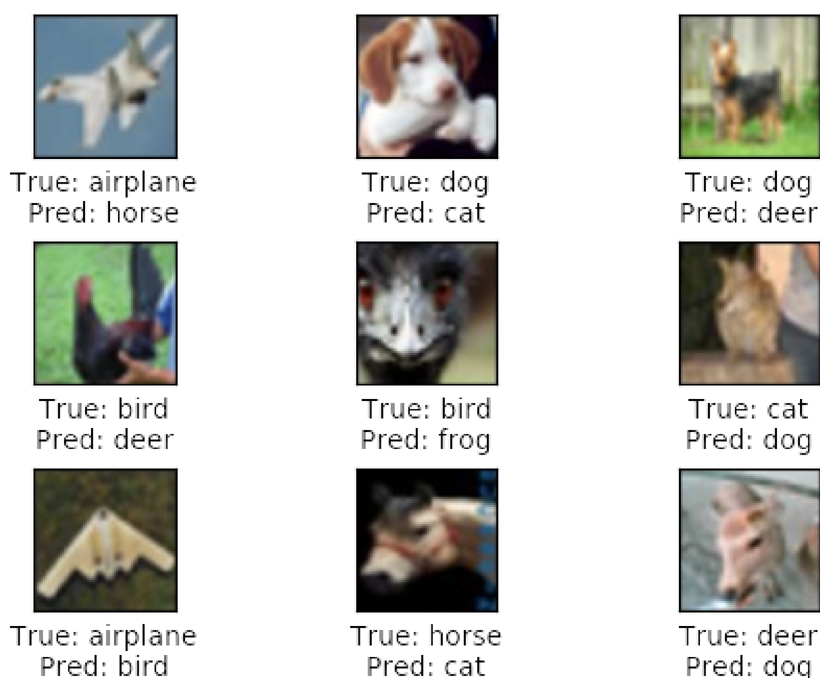
```
if False:
    optimize(num_iterations=1000)
```

结果

在150,000次优化迭代之后，测试集上的分类准确率大约79%-80%。下面画出了一些误分类的图像。其中有一些即使人眼也很难分辨出来，也有一些是合乎情理的错误，比如大型车和卡车，猫与狗，但有些错误就有点奇怪了。

```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

```
Accuracy on Test-Set: 79.3% (7932 / 10000)
Example errors:
```




```
Confusion Matrix:
[775  20  71   8  14   4  18  10  44  36] (0) airplane
[  7 914   5   0   3   7   9   3  14  38] (1) automobile
[ 32   2 724  28  42  44  94  17   9   8] (2) bird
[ 18   7  48 508  56 209  99  29   7  19] (3) cat
[  4   2  45  25 769  29  75  43   3   5] (4) deer
[  8   6  34  89  35 748  38  32   1   9] (5) dog
[  4   2  18   9  14  14 930   4   2   3] (6) frog
[  6   2  23  18  31  55  17 833   0  15] (7) horse
[ 31  29  15  11   8   7  15   0 856  28] (8) ship
[ 13  67   4   5   0   4   7   7  18 875] (9) truck
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

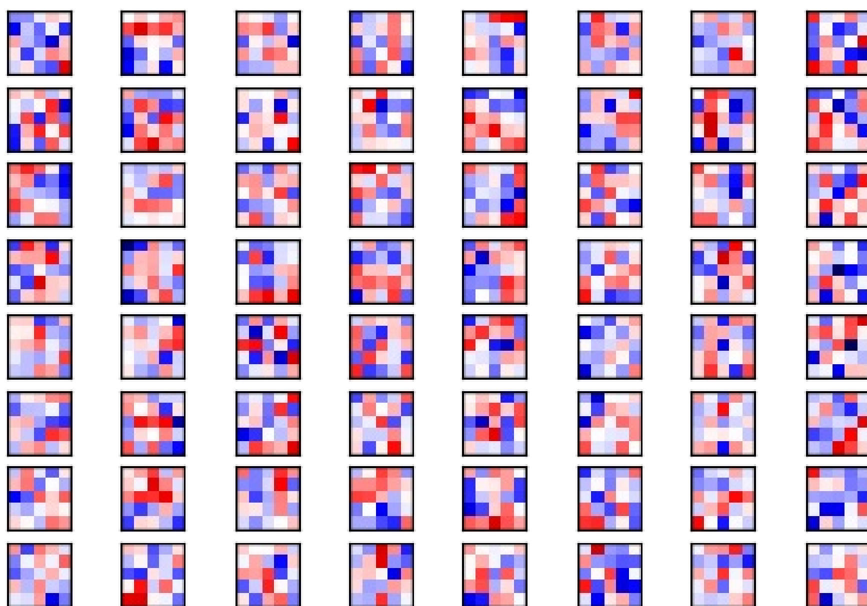
卷积权重

下面展示了一些第一个卷积层的权重（或滤波）。共有3个输入通道，因此有三组（数据），你可以改变 `input_channel` 来改变绘制结果。

权重正值是红的，负值是蓝的。

```
plot_conv_weights(weights=weights_conv1, input_channel=0)
```

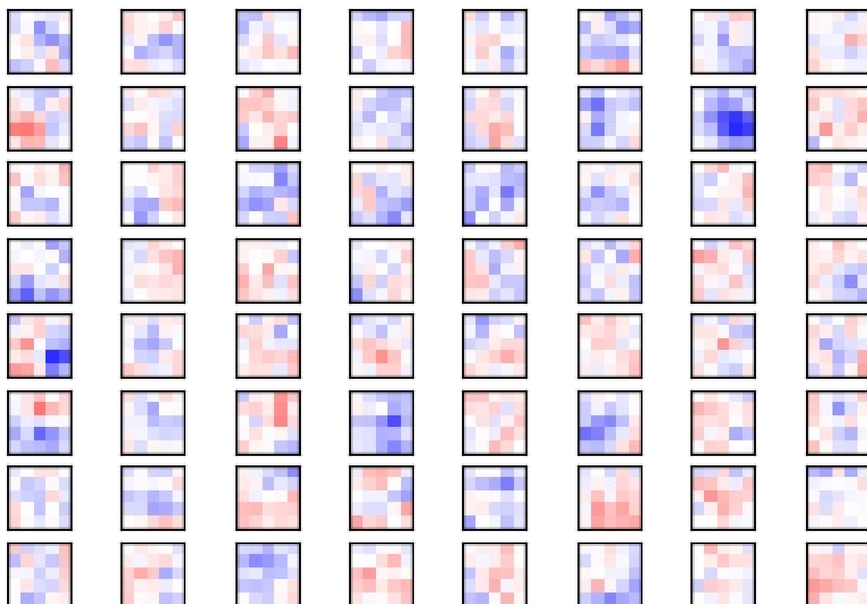
```
Min:   -0.61643, Max:   0.63949
Mean: -0.00177, Stdev: 0.16469
```



下面展示了一些第二个卷积层的权重（或滤波）。它们比第一个卷积层的权重更接近零，你可以看到比较低的标准差。

```
plot_conv_weights(weights=weights_conv2, input_channel=1)
```

```
Min:  -0.73326, Max:   0.25344
Mean: -0.00394, Stdev: 0.05466
```



卷积层的输出

绘制图像的帮助函数。

```
def plot_image(image):
    # Create figure with sub-plots.
    fig, axes = plt.subplots(1, 2)

    # References to the sub-plots.
    ax0 = axes.flat[0]
    ax1 = axes.flat[1]

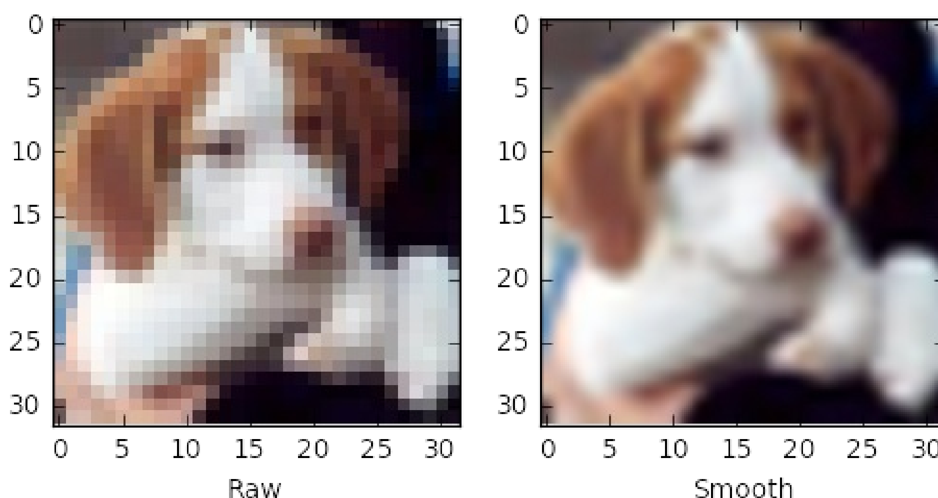
    # Show raw and smoothened images in sub-plots.
    ax0.imshow(image, interpolation='nearest')
    ax1.imshow(image, interpolation='spline16')

    # Set labels.
    ax0.set_xlabel('Raw')
    ax1.set_xlabel('Smooth')

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

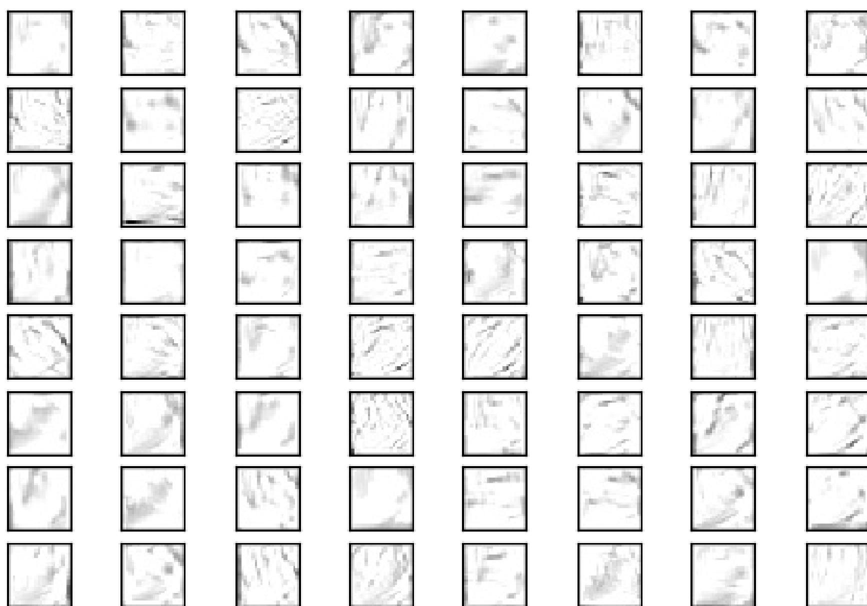
绘制一张测试集中的图像。未处理的像素图像作为神经网络的输入。

```
img, cls = get_test_image(16)
plot_image(img)
```



将原始图像作为神经网络的输入，然后画出第一个卷积层的输出。

```
plot_layer_output(output_conv1, image=img)
```



将同样的图像作为输入，画出第二个卷积层的输出。

```
plot_layer_output(output_conv2, image=img)
```



预测的类别标签

获取图像的预测类别标签和类别号。

```
label_pred, cls_pred = session.run([y_pred, y_pred_cls],
                                   feed_dict={x: [img]})
```

打印预测类别标签。

```
# Set the rounding options for numpy.
np.set_printoptions(precision=3, suppress=True)

# Print the predicted label.
print(label_pred[0])
```

```
[ 0.     0.     0.     0.493  0.     0.49   0.006  0.01   0.
  0.   ]
```

预测类别标签是长度为10的数组，每个元素代表着神经网络有多大信心认为图像是该类别。

在这个例子中，索引3的值是0.493，5的值为0.490。这表示神经网络相信图像要么是类别3，要么是类别5，即猫或狗。

```
class_names[3]
```

```
'cat'
```

```
class_names[5]
```

```
'dog'
```

关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# session.close()
```

结论

这篇教程介绍了如何创建一个在CIRAR-10数据集上进行图像分类的卷积神经网络。测试集上的分类准确率大概79-80%。

同时也画出了卷积层的输出，但很难看出神经网络如何分辨并分类图像。需要更好的可视化技巧。

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Nootbook进行改变之前，可能需要先备份一下。

- 执行10,000次迭代，看看分类准确率如何。将会保存一个checkpoint来储存TensorFlow图的所有变量。
- 再执行100,000次迭代，看看分类准确率有没有提升。然后再执行100,000次。准确率有提升吗，你认为值得这些增加的计算时间吗？
- 试着再预处理阶段改变图像的变体。
- 试着改变神经网络的结构。你可以让神经网络更大或更小。这对训练时间或分类准确率有什么影响？要注意的是，当你改变了神经网络结构时，就无法重新载入checkpoints了。
- 试着在第二个卷积层使用batch-normalization。也试试在俩个层中都删掉它。
- 研究一些CIFAR-10上的更好的神经网络 ，试着实现它们。
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #07

Inception 模型

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

介绍

这篇教程演示了如何用一个预训练好的深度神经网络Inception v3来进行图像分类。

Inception v3模型在一台配有 8 Tesla K40 GPUs，大概价值$30,000的野兽级计算机上训练了几个星期，因此不可能在一台普通的PC上训练。我们将会下载预训练好的Inception模型，然后用它来做图像分类。

Inception v3模型大约有2500万个参数，分类一张图像就用了50亿的乘加指令。在一台没有GPU的现代PC上，分类一张图像转眼就能完成。

这篇教程隐藏了TensorFlow代码，因此可能不要求很多的TensorFlow经验，当然从之前的教程中学到一些对TensorFlow的基本理解还是很有帮助的，特别是在你想学习 `inception.py` 文件中的实现细节时。

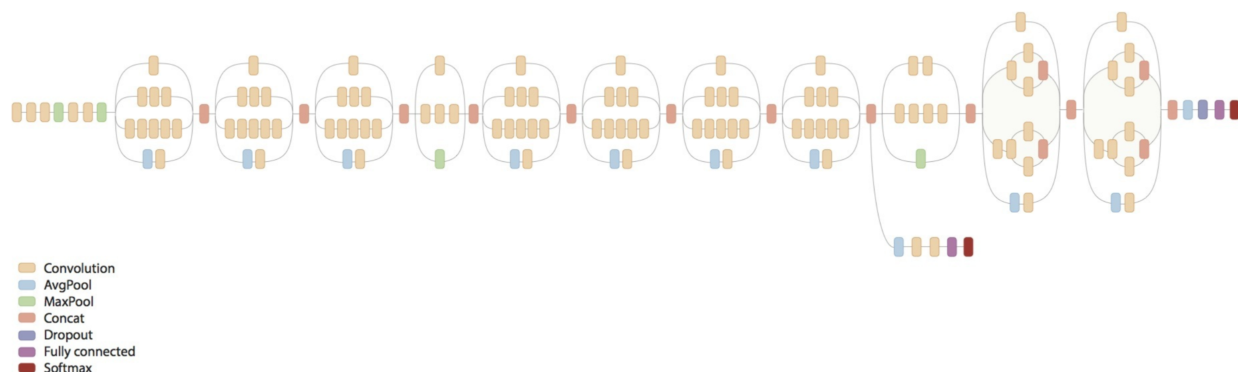
流程图

下面的流程图显示了Inception v3模型中的数据流向，这是一个带有许多层的，有着复杂结构的卷积神经网络。这篇论文里有Inception模型如何构造，以及为什么这么设计的更多细节。但作者也承认他们并不完全明白模型的工作原理。

注意，Inception模型有两个softmax输出。一个是在训练神经网络时使用，另一个是训练结束之后，在图像分类时使用，即推断阶段（inference）。

上周刚刚分布了新的模型， 它比Inception v3更复杂，也得到了更好的分类准确率。

```
from IPython.display import Image, display
Image('images/07_inception_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import os

# Functions and classes for loading and using the Inception mode
l.
import inception
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.10.0rc0'
```

下载Inception模型

从网上下载Inception模型。这是你保存数据文件的默认文件夹。如果文件夹不存在就自动创建。

```
# inception.data_dir = 'inception/'
```

如果文件夹中不存在Inception模型，就自动下载。
它有85MB。

```
inception.maybe_download()
```

```
Downloading Inception v3 Model ...
Data has apparently already been downloaded and unpacked.
```

载入Inception模型

载入模型，为图像分类做准备。

注意这些warning信息，以后可能会导致程序运行失败。

```
model = inception.Inception()
```

```
/home/magnus/anaconda3/envs/tensorflow/lib/python3.5/site-packag
es/tensorflow/python/ops/array_ops.py:1811: VisibleDeprecationWa
rning: converting an array with ndim > 0 to an index will result
 in an error in the future
  result_shape.insert(dim, 1)
```

分类以及绘制图像的帮助函数

这是一个简单的封装函数，它可以展示图像，然后用Inception模型进行分类，最终打印出分类评分。

```
def classify(image_path):
    # Display the image.
    display(Image(image_path))

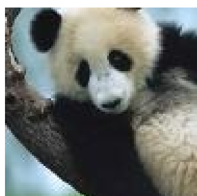
    # Use the Inception model to classify the image.
    pred = model.classify(image_path=image_path)

    # Print the scores and names for the top-10 predictions.
    model.print_scores(pred=pred, k=10, only_first_name=True)
```

熊猫

Inception数据文件中包含了这张熊猫图像。Inception模型相当确定这张图片上展示了熊猫，分类评分达到了89.23%，第二高的代表大狐猴的分数只有0.86%，这是另外一种外来动物。

```
image_path = os.path.join(inception.data_dir, 'cropped_panda.jpg'
)
classify(image_path)
```



```
89.23% : giant panda
 0.86% : indri
 0.26% : lesser panda
 0.14% : custard apple
 0.11% : earthstar
 0.08% : sea urchin
 0.05% : forklift
 0.05% : soccer ball
 0.05% : go-kart
 0.05% : digital watch
```

分类评分的解释

Inception模型的输出是Softmax函数，这在之前教程中的神经网络中也有用到。

softmax输出有时也称为概率分布（probabilities），因为它介于零到一之间，然后相加为一，与概率分布相同。但它们并不是传统语义上的概率分布，因为并不是由重复试验得来。

将神经网络的输出值称为分类评分或排名可能会更好，因为结果显示了神经网络认为输入图像是每个可能分类的强度。

在上面的熊猫样本中， Inception模型给熊猫类型很高的分数—89.23%，同时其它999种类别的分数都在1%以下。这表示Inception模型十分确信图像展示了一只熊猫，而剩下1%以下的应该视为噪声 。比如，排名第十高的分数是0.05%，代表电子手表，但它更可能是由于神经网络的不精准而不是暗示着图像看起来有点像电子手表。

有时Inception模型不确定图像属于哪一个分类，因此结果中并没有一个特别高的分数。下面会展示这种样本。

鹦鹉（原始图像）

Inception模型十分确定（评分97.30%）这张图像展示了一种叫金刚鹦鹉的鹦鹉。

```
classify(image_path="images/parrot.jpg")
```



```
97.30% : macaw
 0.07% : African grey
 0.07% : toucan
 0.05% : jacamar
 0.04% : bee eater
 0.04% : lorikeet
 0.02% : sulphur-crested cockatoo
 0.02% : jay
 0.01% : kite
 0.01% : sandbar
```

鹦鹉（调整图像）

Inception使用于299 x 299像素的输入图像。上面的鹦鹉图像实际上是320像素宽、785像素高的，因此它将由Inception模型自动缩放。

现在我们想看看被Inception模型调整过的图像。

首先我们实现一个帮助函数，用来从Inception模型内部获取调整过的图像。

```
def plot_resized_image(image_path):
    # Get the resized image from the Inception model.
    resized_image = model.get_resized_image(image_path=image_pat
h)

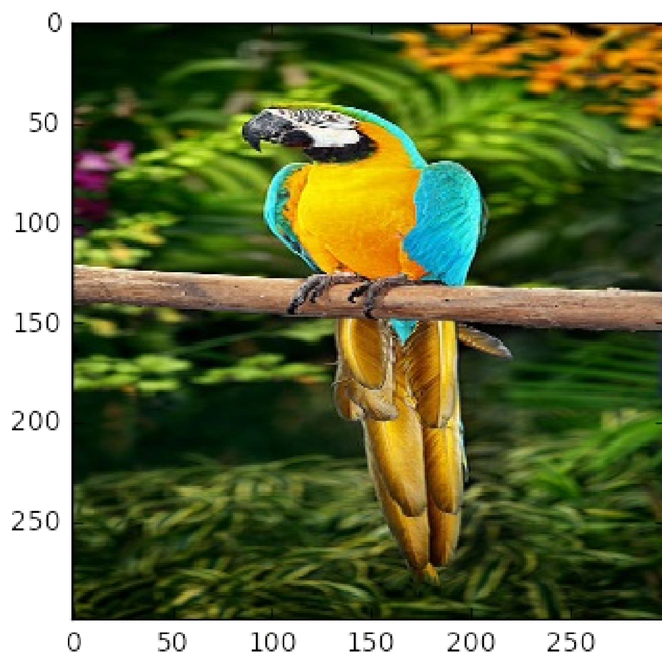
    # Plot the image.
    plt.imshow(resized_image, interpolation='nearest')

    # Ensure that the plot is shown.
    plt.show()
```

现在画出调整过的鹦鹉图。这是Inception模型中神经网络的真正输入图像。我们可以看到它被压缩成正方形，并且分辨率降低了，因此图像看起来更像素化和锯齿状。

这种情况下，图像仍然清晰地展示了一只鹦鹉，但一些图像经过（模型内）原生的调整后会变得扭曲，因此你可能会想自己调整图像大小，再输入到Inception模型。

```
plot_resized_image(image_path="images/parrot.jpg")
```



鹦鹉（裁剪图像，上方）

鹦鹉图像被手动裁剪成299 x 299像素大小，然后输入到Inception模型中，这时（模型）还是很确信（评分97.38%）输入图展示了一只鹦鹉（金刚鹦鹉）。

```
classify(image_path="images/parrot_cropped1.jpg")
```

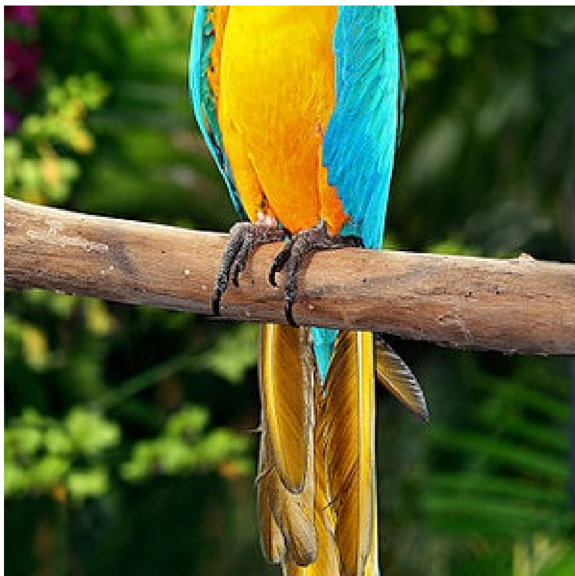


```
97.38% : macaw
 0.09% : African grey
 0.03% : sulphur-crested cockatoo
 0.02% : toucan
 0.02% : reflex camera
 0.01% : comic book
 0.01% : backpack
 0.01% : bib
 0.01% : vulture
 0.01% : lens cap
```

鹦鹉（裁剪图像，中间）

这是鹦鹉图的另一张裁剪图像，这次展示了鹦鹉的躯干，不包含头部和尾巴。Inception模型仍然很确定（评分93.94%）这是一只金刚鹦鹉。

```
classify(image_path="images/parrot_cropped2.jpg")
```



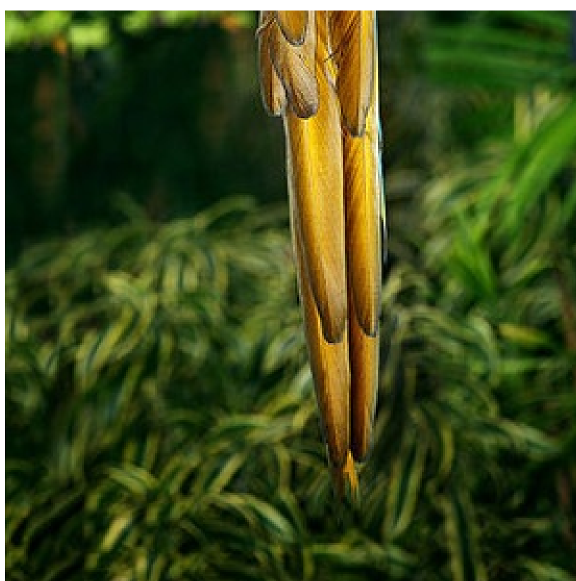
```
93.94% : macaw
 0.77% : toucan
 0.55% : African grey
 0.13% : jacamar
 0.12% : bee eater
 0.11% : sulphur-crested cockatoo
 0.10% : magpie
 0.09% : jay
 0.07% : lorikeet
 0.05% : hornbill
```

鹦鹉（裁剪图像，底部）

这次的裁剪图像只显示了鹦鹉的尾巴。现在Inception模型相当困惑，认为图像可能显示的是一只鵎鵼（评分26.11%），这是另一种外来鸟，也可能是一只草蜢（评分10.61%）。

Inception模型还认为图像有可能是一只钢笔（评分2%）。但这是一个很低的分数，应该解释成不可靠的噪声。

```
classify(image_path="images/parrot_cropped3.jpg")
```



```
26.11% : jacamar
10.61% : grasshopper
 4.05% : chime
 2.24% : bulbul
 2.00% : fountain pen
 1.60% : leafhopper
 1.26% : cricket
 1.25% : kite
 1.13% : macaw
 0.80% : torch
```

鹦鹉（填充图像）

对Inception模型来说，最好的输入图像方式是先将图像填充成正方形，然后调整至299 x 299像素，在这样的鹦鹉样本中，模型正确分类并且评分达到96.78%。

```
classify(image_path="images/parrot_padded.jpg")
```



```
96.78% : macaw
 0.06% : toucan
 0.06% : African grey
 0.05% : bee eater
 0.04% : sulphur-crested cockatoo
 0.03% : king penguin
 0.03% : jacamar
 0.03% : lorikeet
 0.01% : kite
 0.01% : anemone fish
```

Elon Musk (299 x 299 像素)

这张图像展示了Elon Musk——活着的传奇，超级-书呆子-英雄。但Inception模型对图像显示的东西很困惑，它预测图像可能是一件运动衫（评分19.73%），或者是一件阿拉伯长袍（评分16.82）。它也认为图像可能是一个乒乓球（3.05%）或一个棒球（评分1.86%）。Inception模型很困惑，并且分类评分不可靠。

```
classify(image_path="images/elon_musk.jpg")
```



```
19.73% : sweatshirt
16.82% : abaya
 4.17% : suit
 3.46% : trench coat
 3.05% : ping-pong ball
 1.92% : cellular telephone
 1.86% : baseball
 1.77% : jersey
 1.54% : kimono
 1.43% : water bottle
```

Elon Musk (100 x 100 像素)

如果我们使用100 x 100像素的Elon Mask图像，这时Inception模型认为它可能是一件运动衫（评分17.85%），或是一只牛仔靴（评分16.36%）。现在Inception模型做出了一些不同的预测，但仍然很困惑。

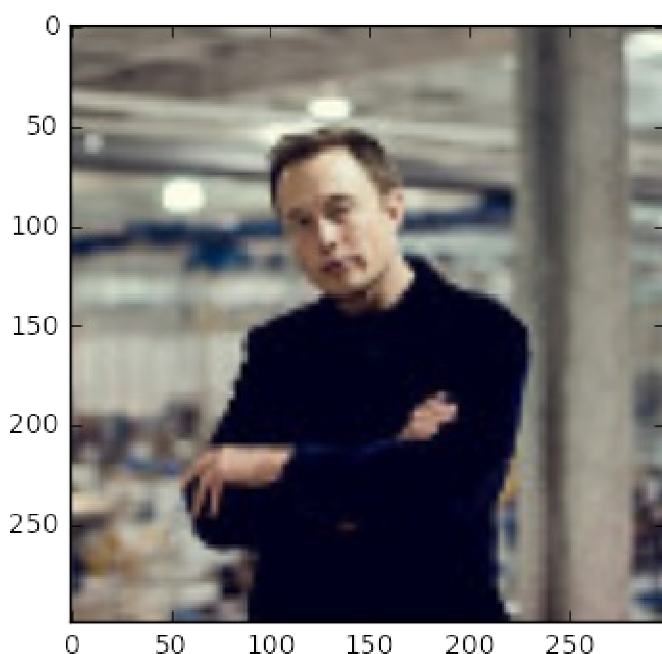
```
classify(image_path="images/elon_musk_100x100.jpg")
```



```
17.85% : sweatshirt
16.36% : cowboy boot
10.68% : balance beam
 8.87% : abaya
 5.36% : suit
 4.57% : Loafer
 2.94% : trench coat
 2.65% : maillot
 1.87% : jersey
 1.42% : unicycle
```

Inception模型自动将图像从100 x 100放大至299 x 299像素，如下所示。看看它是多么像素化和锯齿状，尽管人类可以轻易地看出这是一张双臂交叉的男人的图像。

```
plot_resized_image(image_path="images/elon_musk_100x100.jpg")
```



查理和巧克力工厂 (Gene Wilder)

这张图像展示了在1971版电影《查理和巧克力工厂》中演员Gene Wilder饰演的角色。Inception模型很确定图像显示了一个蝴蝶领结（评分97.22%），尽管这是对的，但人类很可能说这张图片展示的是一个人。

原因可能是Inception模型在训练时将戴蝴蝶结的人的图像分类成蝴蝶领结而不是一个人。因此，或许类别名称应该改成“戴蝴蝶领结的人”而不只是“蝴蝶领结”。

```
classify(image_path="images/willy_wonka_old.jpg")
```



```
97.22% : bow tie
 0.92% : cowboy hat
 0.21% : sombrero
 0.09% : suit
 0.06% : bolo tie
 0.05% : Windsor tie
 0.04% : cornet
 0.03% : flute
 0.02% : banjo
 0.02% : revolver
```

查理和巧克力工厂 (Johnny Depp)

这张图像展示了在2005版电影《查理和巧克力工厂》中演员Johnny Depp饰演的角色。Inception模型认为图像是“太阳镜（sunglasses）”（评分31.48%）或“太阳镜（sunglass）”（评分18.77）。实际上，第一个类别的全名是“太阳镜，深色眼镜，墨镜”。出于某些原因，Inception模型被训练成可以识别两种相似的眼镜。再一次，图像显示了太阳镜，这个结果是对的，但人类很可能会说图像展示的是一个人。

```
classify(image_path="images/willy_wonka_new.jpg")
```



```
31.48% : sunglasses
18.77% : sunglass
 1.55% : velvet
 1.02% : wig
 0.77% : cowboy hat
 0.69% : seat belt
 0.67% : sombrero
 0.62% : jean
 0.46% : poncho
 0.43% : jersey
```

关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。注意，TensorFlow-session是在模型内部的，因此我们通过模型来关闭它。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# model.close()
```

总结

本教程说明了如何使用预训练的Inception v3模型。它在一台野兽级电脑上花了好几周才训练好。但我们可以从网上下载完成的模型，然后在一台普通PC上用它来做图像分类。

不幸的是，Inception模型对识别人物很有问题。这可能是所使用训练集的原因。新版的Inception模型也已经发布了，但它可能也是在同样的训练集上训练，对于识别人物还是有问题。希望未来的模型会训练来识别常见的物体，比如人类。

这篇教程中我们在 `inception.py` 文件中隐藏了TensorFlow的实现细节，因为它有点凌乱，我们可能在之后的教程中仍然会复用这个。希望TensorFlow的开发者会标准化、简单化API，使得更简单地载入这些预训练模型，这样，每个人只需要几行代码就能使用一个强大的图像分类器。

练习

下面是一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行改变之前，可能需要先备份一下。

- 使用你自己的、或者在网上找到的图像。
- 裁剪，调整大小，扭曲图像，看看它如何影响分类准确率。
- 在代码的几个不同地方加入打印信息。你也可以直接运行调试 `inception.py` 文件。
- 试着使用这些刚发布的新模型。它们的载入方式与Inception v3模型不同，实现起来可能更具挑战性。
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #08

迁移学习

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

简介

在前一篇教程 #07 中，我们了解了如何用预训练的Indeption模型来做图像分类。不幸的是，Inception模型似乎无法对人物图像做分类。原因在于该模型所使用的训练集，其中有一些易混淆的类别标签。

Inception模型实际上能够从图像中提取出有用的信息。因此我们可以用其它数据集来训练Inception模型。但如果要在新的数据集上训练这样的模型，需要在一台强大又昂贵的电脑上花费好几周的时间。

相反，我们可以复用预训练的Inception模型，然后只需要替换掉最后做分类的那一层。这个方法叫迁移学习。

本文基于上一篇教程，你需要熟悉教程#07中的Inception模型，以及之前教程中关于如何在TensorFlow中创建和训练神经网络的部分。 这篇教程的部分代码在 `inception.py` 文件中。

流程图

下图展示了用Inception模型做迁移学习时数据的流向。首先，我们在Inception模型中输入并处理一张图像。在模型最终的分类层之前，将所谓的Transfer- Values保存到缓存文件中。

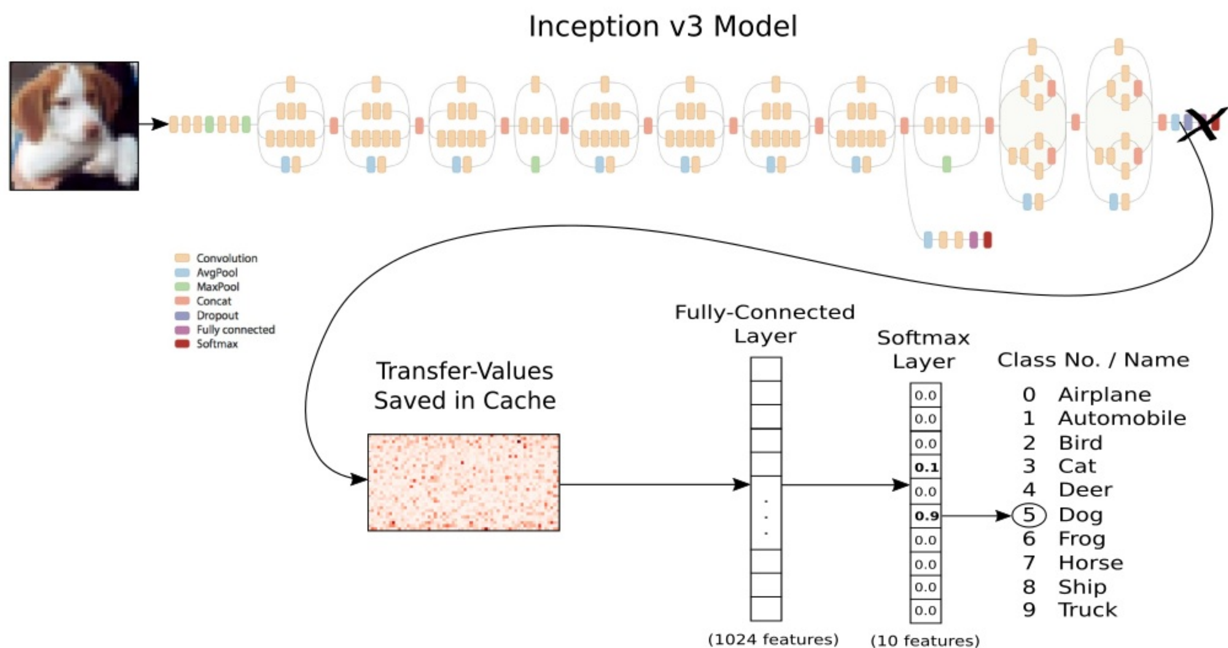
使用缓存文件的原因是，Inception模型处理一张图要花很长时间。我的装有Quad-Core 2 GHz CPU的笔记本电脑每秒能用Inception模型处理3张图像。如果每张图像都要处理多次的话，将transfer-values保存下来可以节省很多时间。

transfer-values有时也称为bottleneck-values，但这个词可能令人费解，在这里就没有使用。

当新数据集里的所有图像都用Inception处理过，并且生成的transfer-values都保存到缓存文件之后，我们可以将这些transfer-values作为其它神经网络的输入。接着训练第二个神经网络，用来分类新的数据集，因此，网络基于Inception模型的transfer-values来学习如何分类图像。

这样，Inception模型从图像中提取出有用的信息，然后用另外的神经网络来做真正的分类工作。

```
from IPython.display import Image, display
Image('images/08_transfer_learning_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import time
from datetime import timedelta
import os

# Functions and classes for loading and using the Inception mode
l.
import inception

# We use Pretty Tensor to define the new classifier.
import prettytensor as pt
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.12.0-rc0'
```

PrettyTensor 版本:

```
pt.__version__
```

```
'0.7.1'
```

载入CIFAR-10数据

```
import cifar10
```

cirfa10模块中已经定义好了数据维度，因此我们需要时只要导入就行。

```
from cifar10 import num_classes
```

设置电脑上保存数据集的路径。

```
# cifar10.data_path = "data/CIFAR-10/"
```

CIFAR-10数据集大概有163MB，如果给定路径没有找到文件的话，将会自动下载。

```
cifar10.maybe_download_and_extract()
```

```
Data has apparently already been downloaded and unpacked.
```

载入类别名称。

```
class_names = cifar10.load_class_names()
class_names
```

```
Loading data: data/CIFAR-10/cifar-10-batches-py/batches.meta
```

```
['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']
```

载入训练集。这个函数返回图像、整形分类号码、以及用One-Hot编码的分类号数组，称为标签。

```
images_train, cls_train, labels_train = cifar10.load_training_da
ta()
```

```
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_1
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_2
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_3
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_4
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_5
```

载入测试集。

```
images_test, cls_test, labels_test = cifar10.load_test_data()
```

```
Loading data: data/CIFAR-10/cifar-10-batches-py/test_batch
```

现在已经载入了CIFAR-10数据集，它包含60,000张图像以及相关的标签（图像的分类）。数据集被分为两个独立的子集，即训练集和测试集。

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(images_train)))
print("- Test-set:\t\t{}".format(len(images_test)))
```

```
Size of:
- Training-set:		50000
- Test-set:		10000
```

用来绘制图片的帮助函数

这个函数用来在3x3的栅格中画9张图像，然后在每张图像下面写出真实类别和预测类别。

```
def plot_images(images, cls_true, cls_pred=None, smooth=True):

    assert len(images) == len(cls_true)

    # Create figure with sub-plots.
    fig, axes = plt.subplots(3, 3)

    # Adjust vertical spacing.
    if cls_pred is None:
        hspace = 0.3
    else:
        hspace = 0.6
    fig.subplots_adjust(hspace=hspace, wspace=0.3)

    # Interpolation type.
    if smooth:
        interpolation = 'spline16'
    else:
        interpolation = 'nearest'

    for i, ax in enumerate(axes.flat):
        # There may be less than 9 images, ensure it doesn't cra
sh.
        if i < len(images):
            # Plot image.
            ax.imshow(images[i],
                      interpolation=interpolation)

            # Name of the true class.
            cls_true_name = class_names[cls_true[i]]

            # Show true and predicted classes.
            if cls_pred is None:
                xlabel = "True: {0}".format(cls_true_name)
            else:
                # Name of the predicted class.
                cls_pred_name = class_names[cls_pred[i]]

                xlabel = "True: {0}\nPred: {1}".format(cls_true_
name, cls_pred_name)

            # Show the classes as the label on the x-axis.
            ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

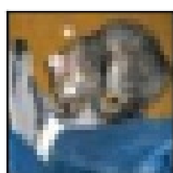
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

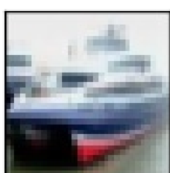
绘制几张图像看看数据是否正确

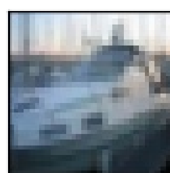
```
# Get the first images from the test-set.
images = images_test[0:9]

# Get the true classes for those images.
cls_true = cls_test[0:9]

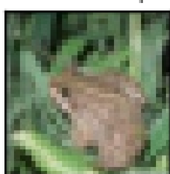
# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true, smooth=False)
```

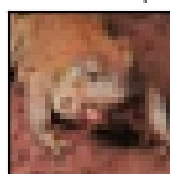

True: cat


True: ship

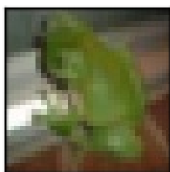

True: ship

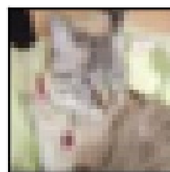

True: airplane


True: frog


True: frog


True: automobile


True: frog


True: cat

下载Inception模型

从网上下载Inception模型。这是你保存数据文件的默认文件夹。如果文件夹不存在就自动创建。

```
# inception.data_dir = 'inception/'
```

如果文件夹中不存在Inception模型，就自动下载。 它有85MB。

更多详情见教程#07。

```
inception.maybe_download()
```

```
Downloading Inception v3 Model ...
Data has apparently already been downloaded and unpacked.
```

载入Inception模型

载入模型，为图像分类做准备。

注意warning信息，以后可能会导致程序运行失败。

```
model = inception.Inception()
```

计算 Transfer-Values

导入用来从Inception模型中获取transfer-values的帮助函数。

```
from inception import transfer_values_cache
```

设置训练集和测试集缓存文件的目录。

```
file_path_cache_train = os.path.join(cifar10.data_path, 'incepti
on_cifar10_train.pkl')
file_path_cache_test = os.path.join(cifar10.data_path, 'inceptio
n_cifar10_test.pkl')
```

```
print("Processing Inception transfer-values for training-images
...")

# Scale images because Inception needs pixels to be between 0 an
d 255,
# while the CIFAR-10 functions return pixels between 0.0 and 1.0
images_scaled = images_train * 255.0

# If transfer-values have already been calculated then reload th
em,
# otherwise calculate them and save them to a cache-file.
transfer_values_train = transfer_values_cache(cache_path=file_pa
th_cache_train,
                                              images=images_scal
ed,
                                              model=model)
```

```
Processing Inception transfer-values for training-images ...
- Data loaded from cache-file: data/CIFAR-10/inception_cifar10_t
rain.pkl
```

```
print("Processing Inception transfer-values for test-images ..."
)

# Scale images because Inception needs pixels to be between 0 an
d 255,
# while the CIFAR-10 functions return pixels between 0.0 and 1.0
images_scaled = images_test * 255.0

# If transfer-values have already been calculated then reload th
em,
# otherwise calculate them and save them to a cache-file.
transfer_values_test = transfer_values_cache(cache_path=file_pat
h_cache_test,
                                             images=images_scale
d,
                                             model=model)
```

```
Processing Inception transfer-values for test-images ...
- Data loaded from cache-file: data/CIFAR-10/inception_cifar10_t
est.pkl
```

检查transfer-values的数组大小。在训练集中有50,000张图像，每张图像有2048个transfer-values。

```
transfer_values_train.shape
```

```
(50000, 2048)
```

相同的，在测试集中有10,000张图像，每张图像有2048个transfer-values。

```
transfer_values_test.shape
```

```
(10000, 2048)
```

绘制transfer-values的帮助函数

```
def plot_transfer_values(i):
    print("Input image:")

    # Plot the i'th image from the test-set.
    plt.imshow(images_test[i], interpolation='nearest')
    plt.show()

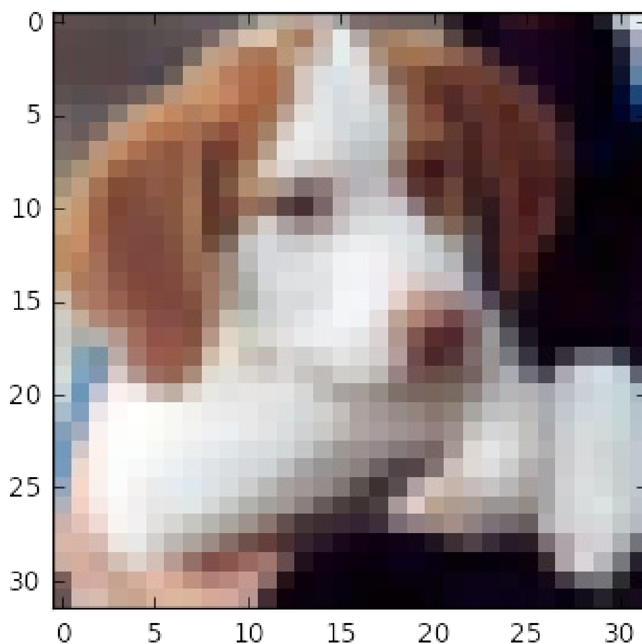
    print("Transfer-values for the image using Inception model:"
)

    # Transform the transfer-values into an image.
    img = transfer_values_test[i]
    img = img.reshape((32, 64))

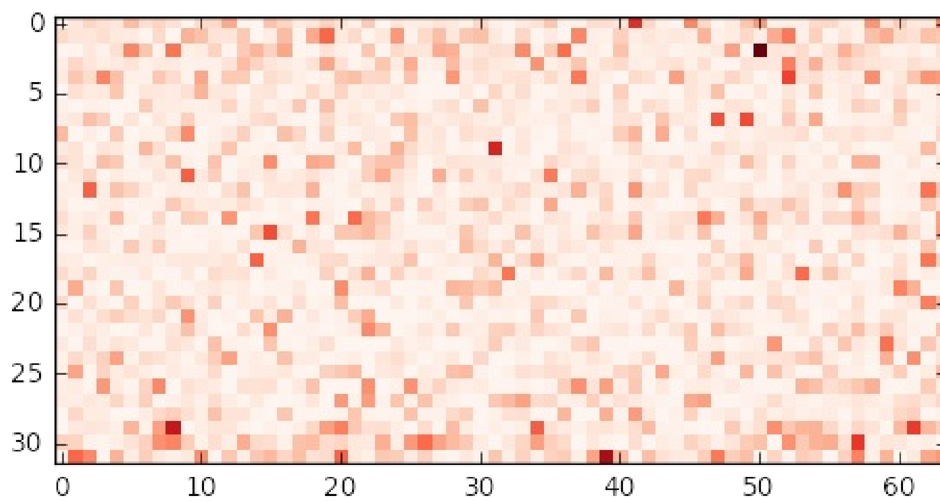
    # Plot the image for the transfer-values.
    plt.imshow(img, interpolation='nearest', cmap='Reds')
    plt.show()
```

```
plot_transfer_values(i=16)
```

```
Input image:
```

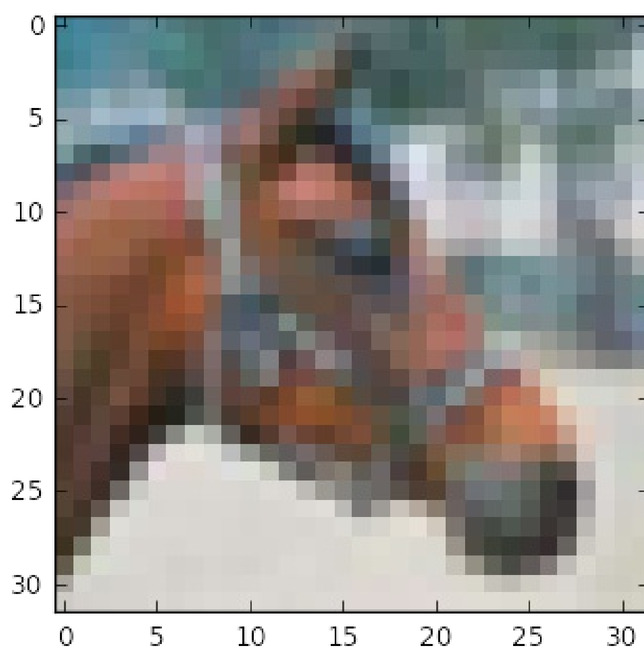



```
Transfer-values for the image using Inception model:
```

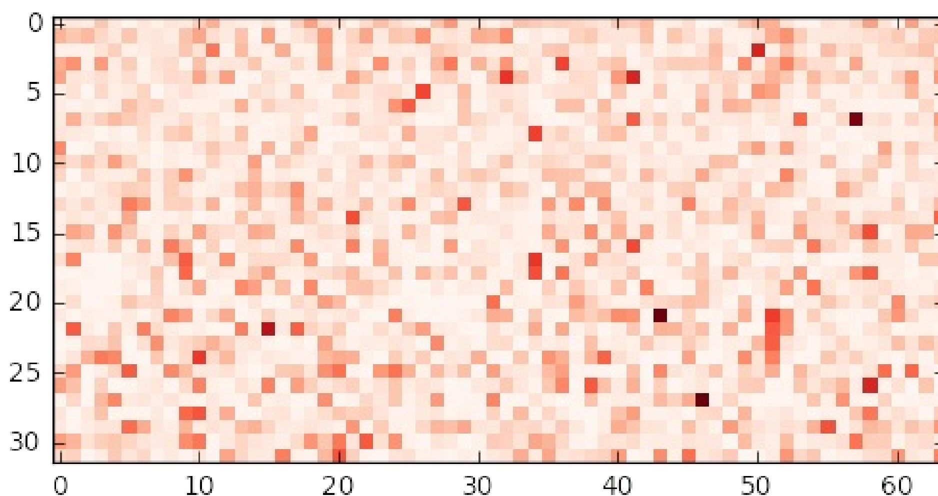



```
plot_transfer_values(i=17)
```

```
Input image:
```




```
Transfer-values for the image using Inception model:
```



transfer-values的PCA分析结果

用scikit-learn里的主成分分析(PCA),将transfer-values的数组维度从2048维降到2维，方便绘制。

```
from sklearn.decomposition import PCA
```

创建一个新的PCA-object，将目标数组维度设为2。

```
pca = PCA(n_components=2)
```

计算PCA需要一段时间，因此将样本数限制在3000。如果你愿意，可以使用整个训练集。

```
transfer_values = transfer_values_train[0:3000]
```

获取你选取的样本的类别号。

```
cls = cls_train[0:3000]
```

保数组有3000份样本,每个样本有2048个transfer-values。

```
transfer_values.shape
```

```
(3000, 2048)
```

用PCA将transfer-value从2048维降低到2维。

```
transfer_values_reduced = pca.fit_transform(transfer_values)
```

数组现在有3000个样本，每个样本两个值。

```
transfer_values_reduced.shape
```

```
(3000, 2)
```

帮助函数用来绘制降维后的transfer-values。

```
def plot_scatter(values, cls):
    # Create a color-map with a different color for each class.
    import matplotlib.cm as cm
    cmap = cm.rainbow(np.linspace(0.0, 1.0, num_classes))

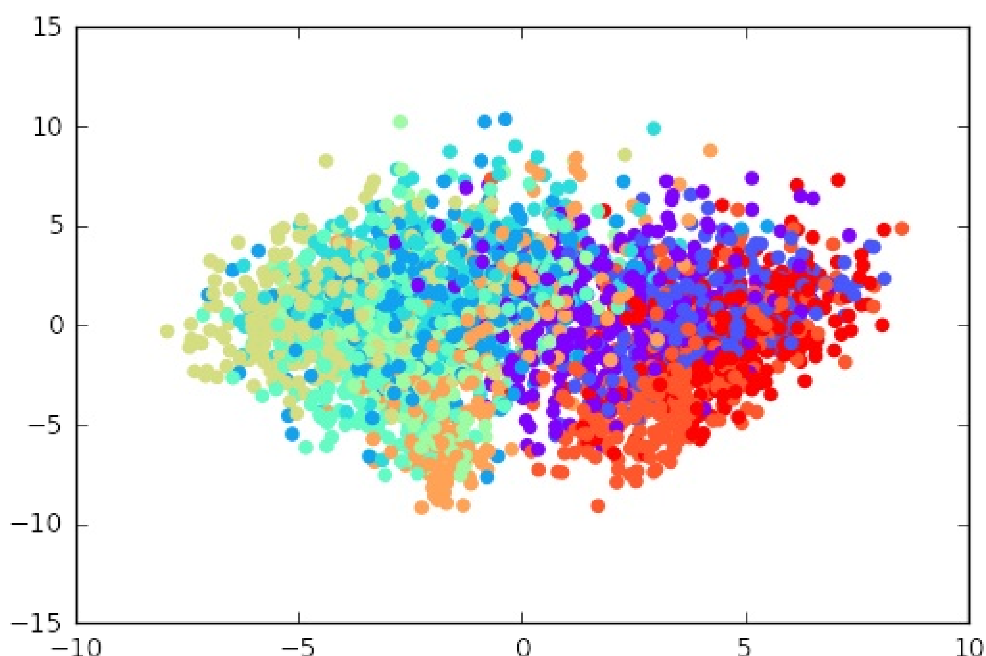
    # Get the color for each sample.
    colors = cmap[cls]

    # Extract the x- and y-values.
    x = values[:, 0]
    y = values[:, 1]

    # Plot it.
    plt.scatter(x, y, color=colors)
    plt.show()
```

画出用PCA降维后的transfer-values。用10种不同的颜色来表示CIFAR-10数据集中不同的类别。颜色各自组合在一起，但有很多重叠部分。这可能是因为PCA无法正确地分离transfer-values。

```
plot_scatter(transfer_values_reduced, cls)
```



transfer-values的t-SNE分析结果

```
from sklearn.manifold import TSNE
```

另一种降维的方法是t-SNE。不幸的是，t-SNE很慢，因此我们先用PCA将维度从2048减少到50。

```
pca = PCA(n_components=50)
transfer_values_50d = pca.fit_transform(transfer_values)
```

创建一个新的t-SNE对象，用来做最后的降维工作，将目标维度设为2维。

```
tsne = TSNE(n_components=2)
```

用t-SNE执行最终的降维。目前在scikit-learn中实现的t-SNE可能无法处理很多样本的数据，所以如果你用整个训练集的话，程序可能会崩溃。

```
transfer_values_reduced = tsne.fit_transform(transfer_values_50d
)
```

确保数组有3000份样本,每个样本有两个transfer-values。

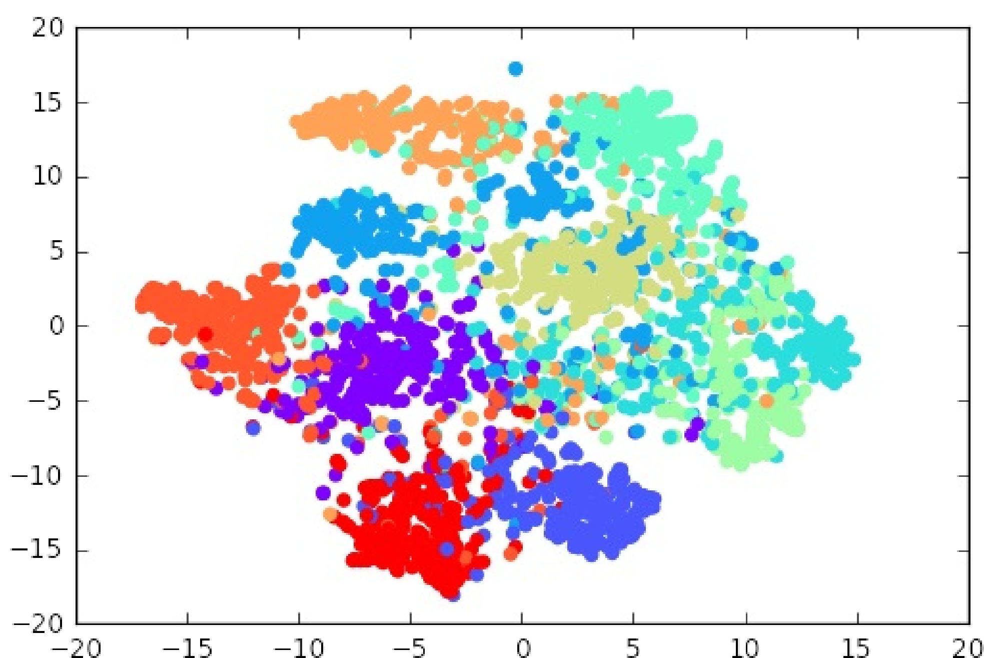
```
transfer_values_reduced.shape
```

```
(3000, 2)
```

画出用t-SNE降低至二维的transfer-values，相比上面PCA的结果，它有更好的分离度。

这意味着由Inception模型得到的transfer-values似乎包含了足够多的信息，可以对CIFAR-10图像进行分类，然而还是有一些重叠部分，说明分离并不完美。

```
plot_scatter(transfer_values_reduced, cls)
```



TensorFlow中的新分类器

在我们将会在TensorFlow中创建一个新的神经网络。这个网络会把Inception模型中的transfer-values作为输入，然后输出CIFAR-10图像的预测类别。

这里假定你已经熟悉如何在TensorFlow中建立神经网络，否则请阅读教程#03。

占位符（Placeholder）变量

首先需要找到transfer-values的数组长度，它是保存在Inception模型对象中的一个变量。

```
transfer_len = model.transfer_len
```

现在为输入的transfer-values创建一个placeholder变量，输入到我们新建的网络中。变量的形状是 `[None, transfer_len]` ， `None` 表示它的输入数组包含任意数量的样本，每个样本元素个数为2048，即 `transfer_len` 。

```
x = tf.placeholder(tf.float32, shape=[None, transfer_len], name=
'x')
```

为输入图像的真实类型标签定义另外一个placeholder变量。这是One-Hot编码的数组，包含10个元素，每个元素代表了数据集中的一种可能类别。

```
y_true = tf.placeholder(tf.float32, shape=[None, num_classes], n
ame='y_true')
```

计算代表真实类别的整形数字。这也可能是一个placeholder变量。

```
y_true_cls = tf.argmax(y_true, dimension=1)
```

神经网络

创建在CIFAR-10数据集上做分类的神经网络。它将Inception模型得到的transfer-values作为输入，保存在placeholder变量 `x` 中。网络输出预测的类别 `y_pred` 。

教程#03中有更多使用Pretty Tensor构造神经网络的细节。

```
# Wrap the transfer-values as a Pretty Tensor object.
x_pretty = pt.wrap(x)

with pt.defaults_scope(activation_fn=tf.nn.relu):
    y_pred, loss = x_pretty.\
        fully_connected(size=1024, name='layer_fc1').\
        softmax_classifier(num_classes=num_classes, labels=y_tru
e)
```

优化方法

创建一个变量来记录当前优化迭代的次数。

```
global_step = tf.Variable(initial_value=0,
                          name='global_step', trainable=False)
```

优化新的神经网络的方法。

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(
loss, global_step)
```

分类准确率

网络的输出y_pred是一个包含10个元素的数组。类别号是数组中最大元素的索引。

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

创建一个布尔向量，表示每张图像的真实类别是否与预测类别相同。

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确度。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
))
```

运行TensorFlow

创建TensorFlow会话（session）

一旦创建了TensorFlow图，我们需要创建一个TensorFlow会话，用来运行图。

```
session = tf.Session()
```

初始化变量

我们需要在开始优化weights和biases变量之前对它们进行初始化。

```
session.run(tf.global_variables_initializer())
```

获取随机训练batch的帮助函数

训练集中有50,000张图像（以及保存transfer-values的数组）。用这些图像（transfer-vlues）计算模型的梯度会花很多时间。因此，我们在优化器的每次迭代里只用到了一小部分的图像（transfer-vlues）。

如果内存耗尽导致电脑死机或变得很慢，你应该试着减少这些数量，但同时可能还需要更优化的迭代。

```
train_batch_size = 64
```

函数用来从训练集中选择随机batch的transfer-vlues。

```
def random_batch():
    # Number of images (transfer-values) in the training-set.
    num_images = len(transfer_values_train)

    # Create a random index.
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)

    # Use the random index to select random x and y-values.
    # We use the transfer-values instead of images as x-values.
    x_batch = transfer_values_train[idx]
    y_batch = labels_train[idx]

    return x_batch, y_batch
```

执行优化迭代的帮助函数

函数用来执行一定数量的优化迭代，以此来逐渐改善网络层的变量。在每次迭代中，会从训练集中选择新的一批数据，然后TensorFlow在这些训练样本上执行优化。每100次迭代会打印出进度。

```
def optimize(num_iterations):
    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(num_iterations):
        # Get a batch of training examples.
        # x_batch now holds a batch of images (transfer-values)
and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = random_batch()

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimiz
er.
        # We also want to retrieve the global_step counter.
        i_global, _ = session.run([global_step, optimizer],
                                  feed_dict=feed_dict_train)

        # Print status to screen every 100 iterations (and last).

        if (i_global % 100 == 0) or (i == num_iterations - 1):
            # Calculate the accuracy on the training-batch.
            batch_acc = session.run(accuracy,
                                    feed_dict=feed_dict_train)

            # Print status.
            msg = "Global Step: {0:>6}, Training Batch Accuracy:
 {1:>6.1%}"
            print(msg.format(i_global, batch_acc))

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_
dif)))))
```

展示结果的帮助函数

绘制错误样本的帮助函数

函数用来绘制测试集中被误分类的样本。

```
def plot_example_errors(cls_pred, correct):
    # This function is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # correct is a boolean array whether the predicted class
    # is equal to the true class for each image in the test-set.

    # Negate the boolean array.
    incorrect = (correct == False)

    # Get the images from the test-set that have been
    # incorrectly classified.
    images = images_test[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = cls_test[incorrect]

    n = min(9, len(images))

    # Plot the first n images.
    plot_images(images=images[0:n],
                cls_true=cls_true[0:n],
                cls_pred=cls_pred[0:n])
```

绘制混淆（confusion）矩阵的帮助函数

```
# Import a function from sklearn to calculate the confusion-matr
ix.
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_test,  # True class for tes
t-set.
                          y_pred=cls_pred)  # Predicted class.

    # Print the confusion matrix as text.
    for i in range(num_classes):
        # Append the class-name to each line.
        class_name = "({}) {}".format(i, class_names[i])
        print(cm[i, :], class_name)

    # Print the class-numbers for easy reference.
    class_numbers = [" ({0})".format(i) for i in range(num_class
es)]
    print("".join(class_numbers))
```

计算分类的帮助函数

这个函数用来计算图像的预测类别，同时返回一个代表每张图像分类是否正确的布尔数组。

由于计算可能会耗费太多内存，就分批处理。如果你的电脑死机了，试着降低batch-size。

```
# Split the data-set in batches of this size to limit RAM usage.
batch_size = 256

def predict_cls(transfer_values, labels, cls_true):
    # Number of images.
    num_images = len(transfer_values)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_images, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing thi
s.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        # Create a feed-dict with the images and labels
        # between index i and j.
        feed_dict = {x: transfer_values[i:j],
                     y_true: labels[i:j]}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_d
ict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Create a boolean array whether each image is correctly cla
ssified.
    correct = (cls_true == cls_pred)

    return correct, cls_pred
```

计算测试集上的预测类别。

```
def predict_cls_test():
    return predict_cls(transfer_values = transfer_values_test,
                       labels = labels_test,
                       cls_true = cls_test)
```

计算分类准确率的帮助函数

这个函数计算了给定布尔数组的分类准确率，布尔数组表示每张图像是否被正确分类。比如，`cls_accuracy([True, True, False, False, False]) = 2/5 = 0.4`。

```
def classification_accuracy(correct):
    # When averaging a boolean array, False means 0 and True means 1.
    # So we are calculating: number of True / len(correct) which is
    # the same as the classification accuracy.

    # Return the classification accuracy
    # and the number of correct classifications.
    return correct.mean(), correct.sum()
```

展示分类准确率的帮助函数

函数用来打印测试集上的分类准确率。

为测试集上的所有图片计算分类会花费一段时间，因此我们直接从这个函数里调用上面的函数，这样就不用每个函数都重新计算分类。

```python
def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # For all the images in the test-set,
    # calculate the predicted classes and whether they are corre
ct.
    correct, cls_pred = predict_cls_test()

    # Classification accuracy and the number of correct classifi
cations.
    acc, num_correct = classification_accuracy(correct)

    # Number of images being classified.
    num_images = len(correct)

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, num_correct, num_images))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

结果

优化之前的性能

测试集上的准确度很低，这是由于模型只做了初始化，并没做任何优化，所以它只是对图像做随机分类。

```python
print_test_accuracy(show_example_errors=False,
                    show_confusion_matrix=False)
```

```
Accuracy on Test-Set: 9.4% (939 / 10000)
```

10,000次优化迭代后的性能

在10,000次优化迭代之后，测试集上的分类准确率大约为90%。相比之下，之前教程#06中的准确率低于80%。

```
optimize(num_iterations=10000)
```

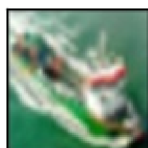
```
Global Step:    100, Training Batch Accuracy:  82.8%
Global Step:    200, Training Batch Accuracy:  90.6%
Global Step:    300, Training Batch Accuracy:  90.6%
Global Step:    400, Training Batch Accuracy:  95.3%
Global Step:    500, Training Batch Accuracy:  85.9%
Global Step:    600, Training Batch Accuracy:  84.4%
Global Step:    700, Training Batch Accuracy:  90.6%
Global Step:    800, Training Batch Accuracy:  93.8%
Global Step:    900, Training Batch Accuracy:  92.2%
Global Step:   1000, Training Batch Accuracy:  95.3%
Global Step:   1100, Training Batch Accuracy:  93.8%
Global Step:   1200, Training Batch Accuracy:  90.6%
Global Step:   1300, Training Batch Accuracy:  95.3%
Global Step:   1400, Training Batch Accuracy:  90.6%
Global Step:   1500, Training Batch Accuracy:  90.6%
Global Step:   1600, Training Batch Accuracy:  92.2%
Global Step:   1700, Training Batch Accuracy:  90.6%
Global Step:   1800, Training Batch Accuracy:  92.2%
Global Step:   1900, Training Batch Accuracy:  84.4%
Global Step:   2000, Training Batch Accuracy:  85.9%
Global Step:   2100, Training Batch Accuracy:  87.5%
Global Step:   2200, Training Batch Accuracy:  90.6%
Global Step:   2300, Training Batch Accuracy:  92.2%
Global Step:   2400, Training Batch Accuracy:  95.3%
Global Step:   2500, Training Batch Accuracy:  89.1%
Global Step:   2600, Training Batch Accuracy:  93.8%
Global Step:   2700, Training Batch Accuracy:  87.5%
Global Step:   2800, Training Batch Accuracy:  90.6%
Global Step:   2900, Training Batch Accuracy:  92.2%
Global Step:   3000, Training Batch Accuracy:  96.9%
Global Step:   3100, Training Batch Accuracy:  96.9%
Global Step:   3200, Training Batch Accuracy:  92.2%
Global Step:   3300, Training Batch Accuracy:  95.3%
Global Step:   3400, Training Batch Accuracy:  93.8%
Global Step:   3500, Training Batch Accuracy:  89.1%
Global Step:   3600, Training Batch Accuracy:  89.1%
Global Step:   3700, Training Batch Accuracy:  95.3%
Global Step:   3800, Training Batch Accuracy:  98.4%
Global Step:   3900, Training Batch Accuracy:  89.1%
Global Step:   4000, Training Batch Accuracy:  92.2%
Global Step:   4100, Training Batch Accuracy:  96.9%
Global Step:   4200, Training Batch Accuracy: 100.0%
Global Step:   4300, Training Batch Accuracy: 100.0%
Global Step:   4400, Training Batch Accuracy:  90.6%
Global Step:   4500, Training Batch Accuracy:  95.3%
```

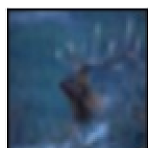
```
Global Step:   4600, Training Batch Accuracy:  96.9%
Global Step:   4700, Training Batch Accuracy:  96.9%
Global Step:   4800, Training Batch Accuracy:  96.9%
Global Step:   4900, Training Batch Accuracy:  92.2%
Global Step:   5000, Training Batch Accuracy:  98.4%
Global Step:   5100, Training Batch Accuracy:  93.8%
Global Step:   5200, Training Batch Accuracy:  92.2%
Global Step:   5300, Training Batch Accuracy:  98.4%
Global Step:   5400, Training Batch Accuracy:  98.4%
Global Step:   5500, Training Batch Accuracy: 100.0%
Global Step:   5600, Training Batch Accuracy:  92.2%
Global Step:   5700, Training Batch Accuracy:  98.4%
Global Step:   5800, Training Batch Accuracy:  92.2%
Global Step:   5900, Training Batch Accuracy:  92.2%
Global Step:   6000, Training Batch Accuracy:  93.8%
Global Step:   6100, Training Batch Accuracy:  95.3%
Global Step:   6200, Training Batch Accuracy:  98.4%
Global Step:   6300, Training Batch Accuracy:  98.4%
Global Step:   6400, Training Batch Accuracy:  96.9%
Global Step:   6500, Training Batch Accuracy:  95.3%
Global Step:   6600, Training Batch Accuracy:  96.9%
Global Step:   6700, Training Batch Accuracy:  96.9%
Global Step:   6800, Training Batch Accuracy:  92.2%
Global Step:   6900, Training Batch Accuracy:  96.9%
Global Step:   7000, Training Batch Accuracy: 100.0%
Global Step:   7100, Training Batch Accuracy:  95.3%
Global Step:   7200, Training Batch Accuracy:  96.9%
Global Step:   7300, Training Batch Accuracy:  96.9%
Global Step:   7400, Training Batch Accuracy:  95.3%
Global Step:   7500, Training Batch Accuracy:  95.3%
Global Step:   7600, Training Batch Accuracy:  93.8%
Global Step:   7700, Training Batch Accuracy:  93.8%
Global Step:   7800, Training Batch Accuracy:  95.3%
Global Step:   7900, Training Batch Accuracy:  95.3%
Global Step:   8000, Training Batch Accuracy:  93.8%
Global Step:   8100, Training Batch Accuracy:  95.3%
Global Step:   8200, Training Batch Accuracy:  98.4%
Global Step:   8300, Training Batch Accuracy:  93.8%
Global Step:   8400, Training Batch Accuracy:  98.4%
Global Step:   8500, Training Batch Accuracy:  96.9%
Global Step:   8600, Training Batch Accuracy:  96.9%
Global Step:   8700, Training Batch Accuracy:  98.4%
Global Step:   8800, Training Batch Accuracy:  95.3%
Global Step:   8900, Training Batch Accuracy:  98.4%
Global Step:   9000, Training Batch Accuracy:  98.4%
Global Step:   9100, Training Batch Accuracy:  98.4%
Global Step:   9200, Training Batch Accuracy:  96.9%
Global Step:   9300, Training Batch Accuracy: 100.0%
Global Step:   9400, Training Batch Accuracy:  90.6%
Global Step:   9500, Training Batch Accuracy:  92.2%
Global Step:   9600, Training Batch Accuracy:  98.4%
Global Step:   9700, Training Batch Accuracy:  96.9%
Global Step:   9800, Training Batch Accuracy:  98.4%
```

```
Global Step:   9900, Training Batch Accuracy:  98.4%
Global Step:  10000, Training Batch Accuracy: 100.0%
Time usage: 0:00:32
```

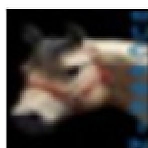
```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

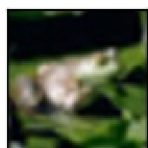
```
Accuracy on Test-Set: 90.7% (9069 / 10000)
Example errors:
```

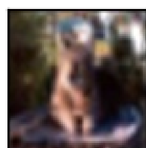

True: ship
Pred: frog

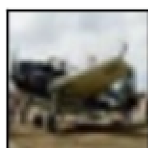

True: deer
Pred: frog

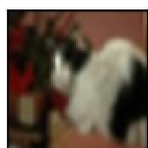

True: automobile
Pred: truck

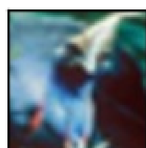

True: horse
Pred: cat


True: frog
Pred: bird


True: cat
Pred: deer


True: airplane
Pred: truck


True: cat
Pred: dog


True: bird
Pred: deer

```
Confusion Matrix:
[926   6  13   2   3   0   1   1  29  19] (0) airplane
[  9 921   2   5   0   1   1   1   2  58] (1) automobile
[ 18   1 883  31  32   4  22   5   1   3] (2) bird
[  7   2  19 855  23  57  24   9   2   2] (3) cat
[  5   0  21  25 896   4  24  22   2   1] (4) deer
[  2   0  12  97  18 843  10  15   1   2] (5) dog
[  2   1  16  17  17   4 940   1   2   0] (6) frog
[  8   0  10  19  28  14   1 914   2   4] (7) horse
[ 42   6   1   4   1   0   2   0 932  12] (8) ship
[  6  19   2   2   1   0   1   1   9 959] (9) truck
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。注意，我们需要关闭两个TensorFlow-session，每个模型对象各有一个。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# model.close()
# session.close()
```

总结

之前的教程 #06 中，我们在一台笔记本电脑上花了15个小时来训练一个神经网络，用来对CIFAR-10数据集做分类，它在测试集上的准确率大约80%。

在这篇教程中，我们使用教程 #07 中的Inception模型，来获取在CIFAR-10数据集上大概90%的分类准确率。我们将所有CIFAR-10数据集中的图像输入到Inception模型中，然后在最终分类层之前获取transfer-values。接着创建另外一个神经网络，它将transfer-values作为输入，生成一个CIFAR-10类别作为输出。

CIFAR-10数据集包含60,000张图像。在一台没有GPU的电脑上，大约花了6个小时来计算Inception模型对这些图像的transfer-values。在这些transfer-values上训练一个新的分类器只需几分钟。两部分时间加起来，这种迁移学习比直接为CIFRA-10数据集训练一个神经网络要快一倍以上，并且它能得到更高的分类准确率。

因此，用Inception模型做迁移学习，对于在自己的数据集上建立一个图像分类器是很有帮助的。

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 试着在PCA和t-SNE中使用整个训练集。会出现什么情况？
- 试着为新的分类器改变神经网络。如果你删掉全连接层或添加更多的全连接层会发生什么？
- 如果你执行更多或更少的迭代会出现什么情况？
- 如果你改变优化器的 `learning_rate` 会发生什么？
- 如果你像在教程#06中的那样，对CIFAR-10图像进行扭曲呢？你将不能使用缓存，因为每张图都不同。
- 试着用MNIST数据集来代替CIFAR-10数据集。

- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #09

视频数据

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

介绍

上一篇教程#08介绍了如何在CIFAR-10数据集上用预训练的Inception模型来做迁移学习（Transfer Learning）。本文将会展示如何使用你自己的图像。

为了示范，我们使用新的数据集Knifey-Spoony，它包含了上千张不同背景下的餐刀、勺子和叉子的图像。训练集有4170张图像，测试集有530张。类别为knifey、sppony和forky，这是对辛普森一家的引用。

knifey-spoony数据集中的图像是用一个简单的Python脚本从视频文件中获取的，脚本在Linux上运行（它需要 `avconv` 程序将视频转成图像）。这让你可以很快地从几分钟的录像视频中创建包含上千张图像的数据集。

本文基于上一篇教程，你需要了解熟悉教程#08中的迁移学习，以及之前教程中关于如何在TensorFlow中创建和训练神经网络的部分。

流程图

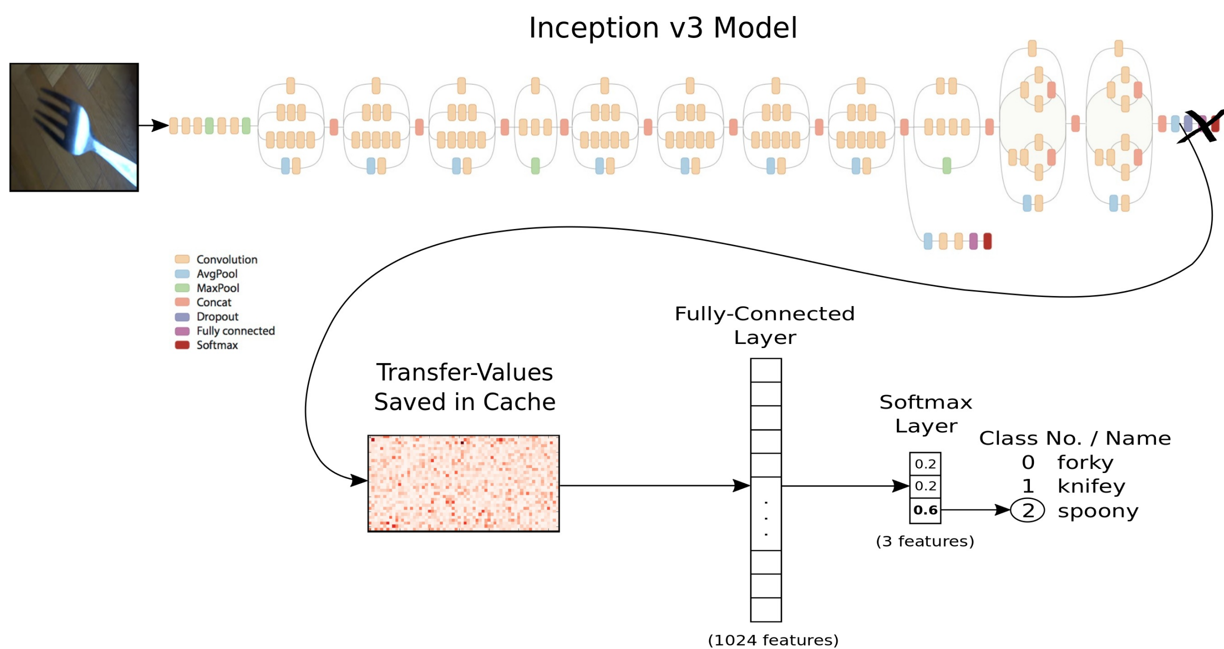
下图展示了用Inception模型做迁移学习时数据的流向。首先，我们在Inception模型中输入并处理一张图像。在模型最终的分类层之前，将所谓的Transfer- Values保存到缓存文件中。

这与在教程#08中做的相似，只是现在用Knifey-Spoony数据集代替CIFAR-10，这说明我们将jpeg图像送到Inception模型中，而不是使用包含图像数据的numpy数组。

当新数据集里的所有图像都用Inception处理过，并且生成的transfer-values都保存到缓存文件之后，我们可以将这些transfer-values作为其它神经网络的输入。接着用新数据集中的类别来训练第二个神经网络，因此，网络基于Inception模型的transfer-values来学习如何分类图像。

这样，Inception模型从图像中提取出有用的信息，然后用另外的神经网络来做真正的分类工作。

```
from IPython.display import Image, display
Image('images/09_transfer_learning_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import time
from datetime import timedelta
import os

# Functions and classes for loading and using the Inception mode
l.
import inception

# We use Pretty Tensor to define the new classifier.
import prettytensor as pt
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.12.0-rc0'
```

PrettyTensor 版本:

```
pt.__version__
```

```
'0.7.1'
```

载入数据

```
import knifey
```

`knifey` 模块中已经定义好了数据维度，因此我们需要时只要导入就行。

```
from knifey import num_classes
```

设置电脑上保存数据集的路径。

```
# knifey.data_dir = "data/knifey-spoony/"
```

设置本教程中保存缓存文件的文件夹。

```
data_dir = knifey.data_dir
```

Knifey-Spoony数据集大概有22MB，如果给定路径没有找到文件的话，将会自动下载。

```
knifey.maybe_download_and_extract()
```

```
Data has apparently already been downloaded and unpacked.
```

现在载入数据集。程序会遍历子文件夹来获取所有 `*.jpg` 格式的图像，然后将文件名放入训练集和测试集的两个列表中。实际上此时并未载入图像，在计算好transfer-values之后再执行载入。

文件名列表将会保存到硬盘上，我们必须确保它们按之后重载数据集的顺序排列。这个很重要，这样我们就能知道哪些图像对应哪些transfer-values。

```
dataset = knifey.load()
```

```
Creating dataset from the files in: data/knifey-spoony/
- Data loaded from cache-file: data/knifey-spoony/knifey-spoony.
pkl
```

你的数据

你可以用自己的图像来代替knifey-spoony数据集。需要创建一个 `dataset.py` 模块中的 `DataSet` 对象。最好的方法是使用 `load_cache()` 封装函数，它会自动将图像列表保存到缓存文件中，因此你需要确保列表顺序和后面生成的transfer-values顺顺序一致。

每个类别的图像需要组织在各自的文件夹里。 `dataset.py` 模块中的文档有更多细节。

```
# This is the code you would run to load your own image-files.
# It has been commented out so it won't run now.

# from dataset import load_cached
# dataset = load_cached(cache_path='my_dataset_cache.pkl', in_di
r='my_images/')
# num_classes = dataset.num_classes
```

训练集和测试集

获取类别名。

```
class_names = dataset.class_names
class_names
```

```
['forky', 'knifey', 'spoony']
```

获取测试集。它返回图像的文件路径、整形类别号和One-Hot编码的类别号数组，称为标签。

```
image_paths_train, cls_train, labels_train = dataset.get_trainin
g_set()
```

打印第一个图像地址，看看是否正确。

```
image_paths_train[0]
```

```
'/home/magnus/development/TensorFlow-Tutorials/data/knifey-spoon
y/forky/forky-05-0023.jpg'
```

获取测试集。

```
image_paths_test, cls_test, labels_test = dataset.get_test_set()
```

打印第一个图像地址，看看是否正确。

```
image_paths_test[0]
```

```
'/home/magnus/development/TensorFlow-Tutorials/data/knifey-spoon
y/forky/test/forky-test-01-0163.jpg'
```

现在已经载入了Knifey-Spoony数据集，它包含4700张图像以及相应的标签（图像的分类）。数据集被手动地分为两个子集，训练集和测试集。

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(image_paths_train)))
print("- Test-set:\t\t{}".format(len(image_paths_test)))
```

```
Size of:
- Training-set:         4170
- Test-set:         530
```

用来绘制图像的帮助函数

这个函数用来在3x3的栅格中画9张图像，然后在每张图像下面写出真实类别和预测类别。

```
def plot_images(images, cls_true, cls_pred=None, smooth=True):

    assert len(images) == len(cls_true)

    # Create figure with sub-plots.
    fig, axes = plt.subplots(3, 3)

    # Adjust vertical spacing.
    if cls_pred is None:
        hspace = 0.3
    else:
        hspace = 0.6
    fig.subplots_adjust(hspace=hspace, wspace=0.3)

    # Interpolation type.
    if smooth:
        interpolation = 'spline16'
    else:
        interpolation = 'nearest'

    for i, ax in enumerate(axes.flat):
        # There may be less than 9 images, ensure it doesn't cra
sh.
        if i < len(images):
            # Plot image.
            ax.imshow(images[i],
                      interpolation=interpolation)

            # Name of the true class.
            cls_true_name = class_names[cls_true[i]]

            # Show true and predicted classes.
            if cls_pred is None:
                xlabel = "True: {0}".format(cls_true_name)
            else:
                # Name of the predicted class.
                cls_pred_name = class_names[cls_pred[i]]

                xlabel = "True: {0}\nPred: {1}".format(cls_true_
name, cls_pred_name)

            # Show the classes as the label on the x-axis.
            ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

载入图像的帮助函数

数据集并未载入实际图像，在训练集和测试集中各有一个图像（地址）列表。下面的帮助函数载入了一些图像文件。

```
from matplotlib.image import imread

def load_images(image_paths):
    # Load the images from disk.
    images = [imread(path) for path in image_paths]

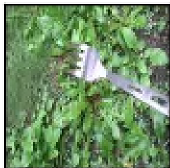
    # Convert to a numpy array and return it.
    return np.asarray(images)
```

绘制一些图像看看数据是否正确

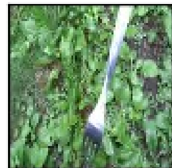
```
# Load the first images from the test-set.
images = load_images(image_paths=image_paths_test[0:9])

# Get the true classes for those images.
cls_true = cls_test[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true, smooth=True)
```



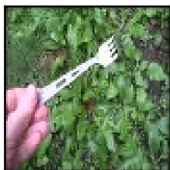
True: forky



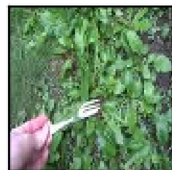
True: forky



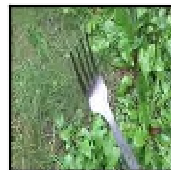
True: forky



True: forky



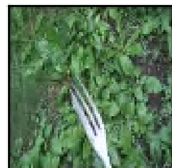
True: forky



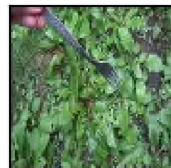
True: forky



True: forky



True: forky



True: forky

下载Inception模型

从网上下载Inception模型。这是你保存数据文件的默认文件夹。如果文件夹不存在就自动创建。

```
# inception.data_dir = 'inception/'
```

如果文件夹中不存在Inception模型，就自动下载。 它有85MB。

更多详情见教程#07。

```
inception.maybe_download()
```

```
Downloading Inception v3 Model ...
Data has apparently already been downloaded and unpacked.
```

载入Inception模型

载入模型，为图像分类做准备。

注意warning信息，以后可能会导致程序运行失败。

```
model = inception.Inception()
```

计算 Transfer-Values

导入用来从Inception模型中获取transfer-values的帮助函数。

```
from inception import transfer_values_cache
```

设置训练集和测试集缓存文件的目录。

```
file_path_cache_train = os.path.join(data_dir, 'inception-knifey
-train.pkl')
file_path_cache_test = os.path.join(data_dir, 'inception-knifey-
test.pkl')
```

```
print("Processing Inception transfer-values for training-images
...")

# If transfer-values have already been calculated then reload th
em,
# otherwise calculate them and save them to a cache-file.
transfer_values_train = transfer_values_cache(cache_path=file_pa
th_cache_train,
                                              image_paths=image_
paths_train,
                                              model=model)
```

```
Processing Inception transfer-values for training-images ...
- Data loaded from cache-file: data/knifey-spoony/inception-knif
ey-train.pkl
```

```
print("Processing Inception transfer-values for test-images ..."
)

# If transfer-values have already been calculated then reload th
em,
# otherwise calculate them and save them to a cache-file.
transfer_values_test = transfer_values_cache(cache_path=file_pat
h_cache_test,
                                             image_paths=image_p
aths_test,
                                             model=model)
```

```
Processing Inception transfer-values for test-images ...
- Data loaded from cache-file: data/knifey-spoony/inception-knif
ey-test.pkl
```

检查transfer-values的数组大小。在训练集中有4170张图像，每张图像有2048个transfer-values。

```
transfer_values_train.shape
```

```
(4170, 2048)
```

同样，在测试集中有530张图像，每张图像有2048个transfer-values。

```
transfer_values_test.shape
```

```
(530, 2048)
```

绘制transfer-values的帮助函数

```
def plot_transfer_values(i):
    print("Input image:")

    # Plot the i'th image from the test-set.
    image = imread(image_paths_test[i])
    plt.imshow(image, interpolation='spline16')
    plt.show()

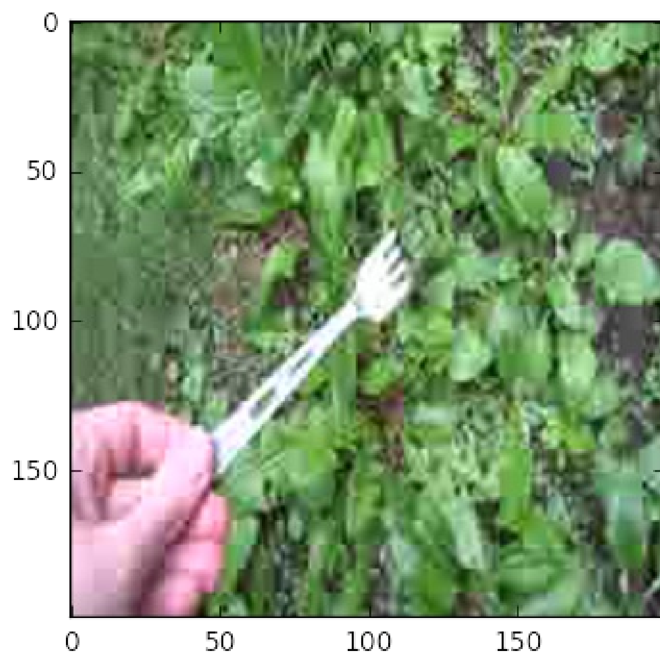
    print("Transfer-values for the image using Inception model:"
)

    # Transform the transfer-values into an image.
    img = transfer_values_test[i]
    img = img.reshape((32, 64))

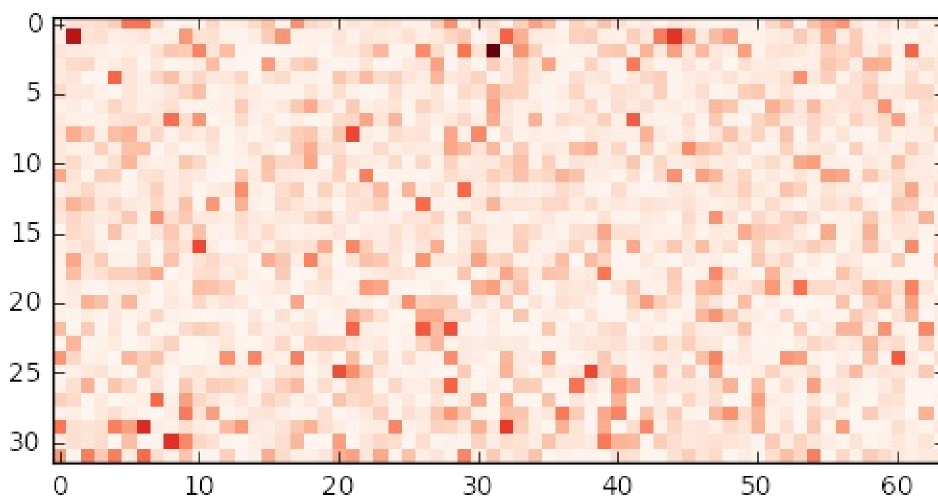
    # Plot the image for the transfer-values.
    plt.imshow(img, interpolation='nearest', cmap='Reds')
    plt.show()
```

```
plot_transfer_values(i=100)
```

```
Input image:
```

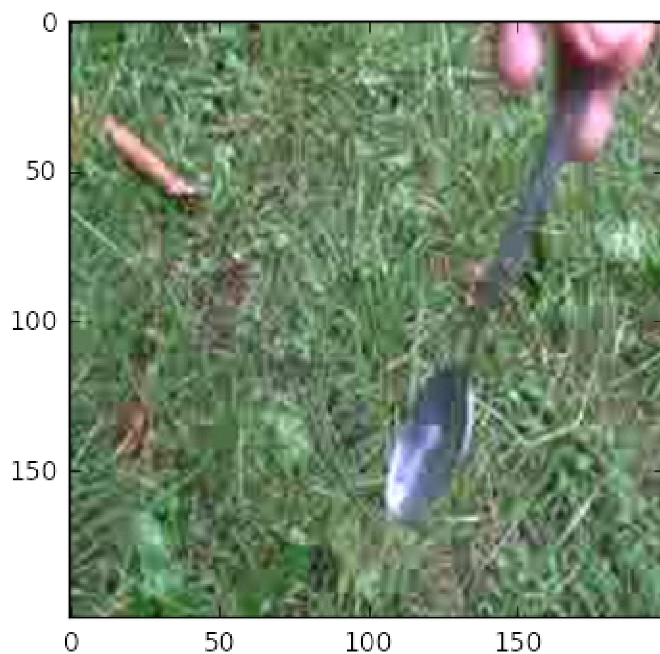


```
Transfer-values for the image using Inception model:
```

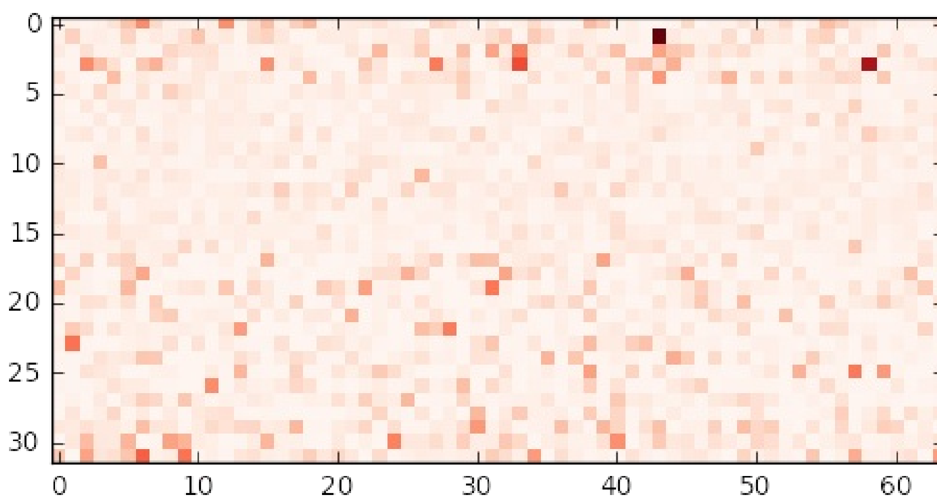


```
plot_transfer_values(i=300)
```

```
Input image:
```




```
Transfer-values for the image using Inception model:
```



transfer-values的PCA分析结果

用scikit-learn里的主成分分析(PCA),将transfer-values的数组维度从2048维降到2维，方便绘制。

```
from sklearn.decomposition import PCA
```

创建一个新的PCA-object，将目标数组维度设为2。

```
pca = PCA(n_components=2)
```

计算PCA需要一段时间。本文的数据集并不是很大，否则你需要挑选训练集中的一小部分，来加速计算。

```
# transfer_values = transfer_values_train[0:3000]
transfer_values = transfer_values_train
```

获取你选取的样本的类别号。

```
# cls = cls_train[0:3000]
cls = cls_train
```

保数组有4170份样本,每个样本有2048个transfer-values。

```
transfer_values.shape
```

```
(4170, 2048)
```

用PCA将transfer-value从2048维降低到2维。

```
transfer_values_reduced = pca.fit_transform(transfer_values)
```

数组现在有4170个样本，每个样本两个值。

```
transfer_values_reduced.shape
```

```
(4170, 2)
```

帮助函数用来绘制降维后的transfer-values。

```
def plot_scatter(values, cls):
    # Create a color-map with a different color for each class.
    import matplotlib.cm as cm
    cmap = cm.rainbow(np.linspace(0.0, 1.0, num_classes))

    # Create an index with a random permutation to make a better
 plot.
    idx = np.random.permutation(len(values))

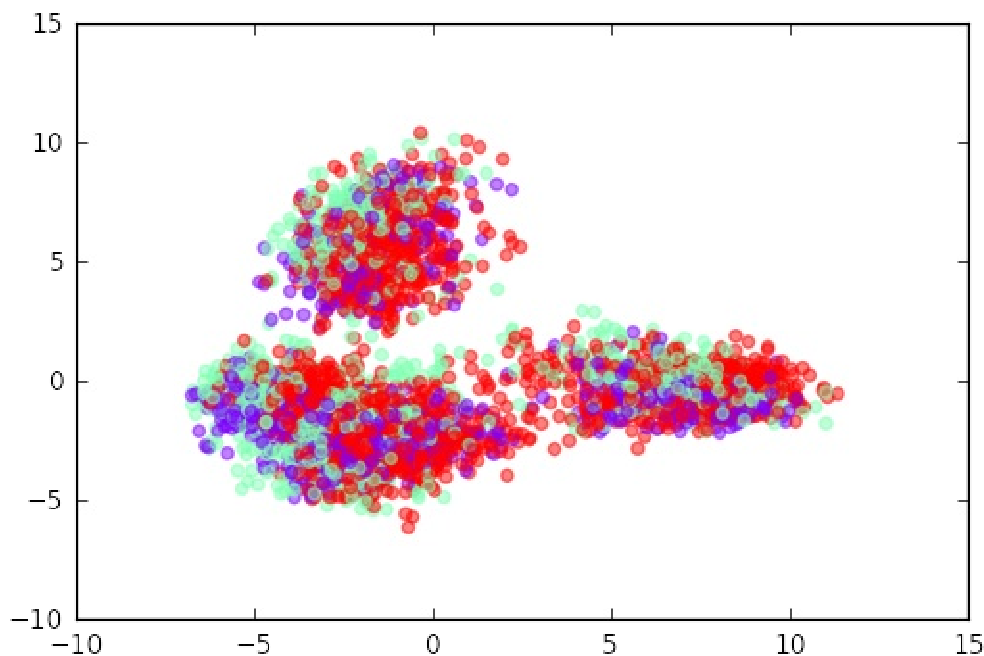
    # Get the color for each sample.
    colors = cmap[cls[idx]]

    # Extract the x- and y-values.
    x = values[idx, 0]
    y = values[idx, 1]

    # Plot it.
    plt.scatter(x, y, color=colors, alpha=0.5)
    plt.show()
```

画出用PCA降维后的transfer-values。用3种不同的颜色来表示Knifey-Spoony数据集中不同的类别。颜色有很多重叠部分。这可能是因为PCA无法正确地分离transfer-values。

```
plot_scatter(transfer_values_reduced, cls=cls)
```



transfer-values的t-SNE分析结果

```
from sklearn.manifold import TSNE
```

另一种降维的方法是t-SNE。不幸的是，t-SNE很慢，因此我们先用PCA将维度从2048减少到50。

```
pca = PCA(n_components=50)
transfer_values_50d = pca.fit_transform(transfer_values)
```

创建一个新的t-SNE对象，用来做最后的降维工作，将目标维度设为2维。

```
tsne = TSNE(n_components=2)
```

用t-SNE执行最终的降维。目前在scikit-learn中实现的t-SNE可能无法处理很多样本的数据，所以如果你用整个训练集的话，程序可能会崩溃。

```
transfer_values_reduced = tsne.fit_transform(transfer_values_50d
)
```

确保数组有4170份样本,每个样本有两个transfer-values。

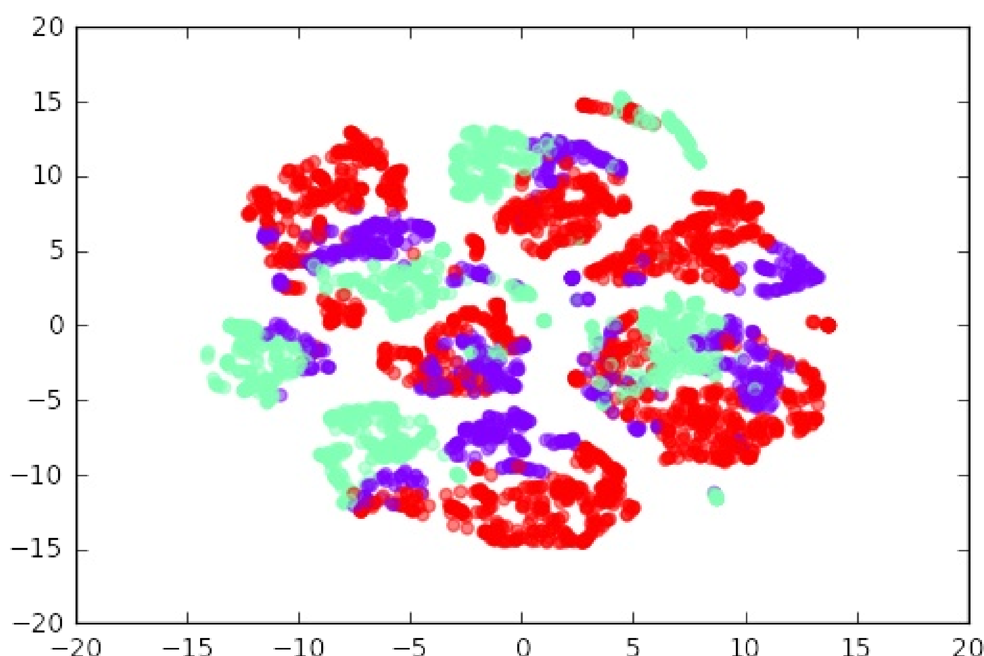
```
transfer_values_reduced.shape
```

```
(4170, 2)
```

画出用t-SNE降低至二维的transfer-values，相比上面PCA的结果，它有更好的分离度。

这意味着由Inception模型得到的transfer-values似乎包含了足够多的信息，可以对Knifey-Sponny图像进行分类，然而还是有一些重叠部分，说明分离并不完美。

```
plot_scatter(transfer_values_reduced, cls=cls)
```



TensorFlow中的新分类器

在我们将会在TensorFlow中创建一个新的神经网络。这个网络会把Inception模型中的transfer-values作为输入，然后输出Knifey-Spoony图像的预测类别。

这里假定你已经熟悉如何在TensorFlow中建立神经网络，否则请阅读教程#03。

占位符（Placeholder）变量

首先需要找到transfer-values的数组长度，它是保存在Inception模型对象中的一个变量。

```
transfer_len = model.transfer_len
```

现在为输入的transfer-values创建一个placeholder变量，输入到我们新建的网络中。变量的形状是 `[None, transfer_len]` ， `None` 表示它的输入数组包含任意数量的样本，每个样本元素个数为2048，即 `transfer_len` 。

```
x = tf.placeholder(tf.float32, shape=[None, transfer_len], name=
'x')
```

为输入图像的真实类型标签定义另外一个placeholder变量。这是One-Hot编码的数组，包含10个元素，每个元素代表了数据集中的一种可能类别。

```
y_true = tf.placeholder(tf.float32, shape=[None, num_classes], n
ame='y_true')
```

计算代表真实类别的整形数字。这也可能是一个placeholder变量。

```
y_true_cls = tf.argmax(y_true, dimension=1)
```

神经网络

创建在Knifey-Spoony数据集上做分类的神经网络。它将Inception模型得到的transfer-values作为输入，保存在placeholder变量 `x` 中。网络输出预测的类别 `y_pred` 。

教程#03中有更多使用Pretty Tensor构造神经网络的细节。

```
# Wrap the transfer-values as a Pretty Tensor object.
x_pretty = pt.wrap(x)

with pt.defaults_scope(activation_fn=tf.nn.relu):
    y_pred, loss = x_pretty.\
        fully_connected(size=1024, name='layer_fc1').\
        softmax_classifier(num_classes=num_classes, labels=y_tru
e)
```

优化方法

创建一个变量来记录当前优化迭代的次数。

```
global_step = tf.Variable(initial_value=0,
                          name='global_step', trainable=False)
```

优化新的神经网络的方法。

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(
loss, global_step)
```

分类准确率

网络的输出 `y_pred` 是一个包含3个元素的数组。类别号是数组中最大元素的索引。

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

创建一个布尔向量，表示每张图像的真实类别是否与预测类别相同。

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确度。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
))
```

运行TensorFlow

创建TensorFlow会话（session）

一旦创建了TensorFlow图，我们需要创建一个TensorFlow会话，用来运行图。

```
session = tf.Session()
```

初始化变量

我们需要在开始优化weights和biases变量之前对它们进行初始化。

```
session.run(tf.global_variables_initializer())
```

获取随机训练batch的帮助函数

训练集中有4170张图像（以及保存transfer-values的数组）。用这些图像（transfer-vlues）计算模型的梯度会花很多时间。因此，我们在优化器的每次迭代里只用到了一小部分的图像（transfer-vlues）。

如果内存耗尽导致电脑死机或变得很慢，你应该试着减少这些数量，但同时可能还需要更优化的迭代。

```
train_batch_size = 64
```

函数用来从训练集中选择随机batch的transfer-vlues。

```python
def random_batch():
    # Number of images (transfer-values) in the training-set.
    num_images = len(transfer_values_train)

    # Create a random index.
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)

    # Use the random index to select random x and y-values.
    # We use the transfer-values instead of images as x-values.
    x_batch = transfer_values_train[idx]
    y_batch = labels_train[idx]

    return x_batch, y_batch
```

执行优化迭代的帮助函数

函数用来执行一定数量的优化迭代，以此来逐渐改善网络层的变量。在每次迭代中，会从训练集中选择新的一批数据，然后TensorFlow在这些训练样本上执行优化。每100次迭代会打印出进度。

```
def optimize(num_iterations):
    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(num_iterations):
        # Get a batch of training examples.
        # x_batch now holds a batch of images (transfer-values)
and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = random_batch()

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimiz
er.
        # We also want to retrieve the global_step counter.
        i_global, _ = session.run([global_step, optimizer],
                                  feed_dict=feed_dict_train)

        # Print status to screen every 100 iterations (and last).

        if (i_global % 100 == 0) or (i == num_iterations - 1):
            # Calculate the accuracy on the training-batch.
            batch_acc = session.run(accuracy,
                                    feed_dict=feed_dict_train)

            # Print status.
            msg = "Global Step: {0:>6}, Training Batch Accuracy:
 {1:>6.1%}"
            print(msg.format(i_global, batch_acc))

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_
dif)))))
```

展示结果的帮助函数

绘制错误样本的帮助函数

函数用来绘制测试集中被误分类的样本。

```
def plot_example_errors(cls_pred, correct):
    # This function is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # correct is a boolean array whether the predicted class
    # is equal to the true class for each image in the test-set.

    # Negate the boolean array.
    incorrect = (correct == False)

    # Get the indices for the incorrectly classified images.
    idx = np.flatnonzero(incorrect)

    # Number of images to select, max 9.
    n = min(len(idx), 9)

    # Randomize and select n indices.
    idx = np.random.choice(idx,
                           size=n,
                           replace=False)

    # Get the predicted classes for those images.
    cls_pred = cls_pred[idx]

    # Get the true classes for those images.
    cls_true = cls_test[idx]

    # Load the corresponding images from the test-set.
    # Note: We cannot do image_paths_test[idx] on lists of strin
gs.
    image_paths = [image_paths_test[i] for i in idx]
    images = load_images(image_paths)

    # Plot the images.
    plot_images(images=images,
                cls_true=cls_true,
                cls_pred=cls_pred)
```

绘制混淆（confusion）矩阵的帮助函数

```
# Import a function from sklearn to calculate the confusion-matr
ix.
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_test,  # True class for tes
t-set.
                          y_pred=cls_pred)  # Predicted class.

    # Print the confusion matrix as text.
    for i in range(num_classes):
        # Append the class-name to each line.
        class_name = "({}) {}".format(i, class_names[i])
        print(cm[i, :], class_name)

    # Print the class-numbers for easy reference.
    class_numbers = [" ({0})".format(i) for i in range(num_class
es)]
    print("".join(class_numbers))
```

计算分类的帮助函数

这个函数用来计算图像的预测类别，同时返回一个代表每张图像分类是否正确的布尔数组。

由于计算可能会耗费太多内存，就分批处理。如果你的电脑死机了，试着降低batch-size。

```
# Split the data-set in batches of this size to limit RAM usage.
batch_size = 256

def predict_cls(transfer_values, labels, cls_true):
    # Number of images.
    num_images = len(transfer_values)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_images, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing thi
s.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        # Create a feed-dict with the images and labels
        # between index i and j.
        feed_dict = {x: transfer_values[i:j],
                     y_true: labels[i:j]}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_d
ict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Create a boolean array whether each image is correctly cla
ssified.
    correct = (cls_true == cls_pred)

    return correct, cls_pred
```

计算测试集上的预测类别。

```
def predict_cls_test():
    return predict_cls(transfer_values = transfer_values_test,
                       labels = labels_test,
                       cls_true = cls_test)
```

计算分类准确率的帮助函数

这个函数计算了给定布尔数组的分类准确率，布尔数组表示每张图像是否被正确分类。比如，
`cls_accuracy([True, True, False, False, False]) = 2/5 = 0.4`。

```
def classification_accuracy(correct):
    # When averaging a boolean array, False means 0 and True mea
ns 1.
    # So we are calculating: number of True / len(correct) which
 is
    # the same as the classification accuracy.

    # Return the classification accuracy
    # and the number of correct classifications.
    return correct.mean(), correct.sum()
```

展示分类准确率的帮助函数

函数用来打印测试集上的分类准确率。

为测试集上的所有图片计算分类会花费一段时间，因此我们直接从这个函数里调用上面的函数，这样就不用每个函数都重新计算分类。

```
def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # For all the images in the test-set,
    # calculate the predicted classes and whether they are corre
ct.
    correct, cls_pred = predict_cls_test()

    # Classification accuracy and the number of correct classifi
cations.
    acc, num_correct = classification_accuracy(correct)

    # Number of images being classified.
    num_images = len(correct)

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, num_correct, num_images))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

结果

优化之前的性能

测试集上的准确度很低，这是由于模型只做了初始化，并没做任何优化，所以它只是对图像做随机分类。

```
print_test_accuracy(show_example_errors=False,
                    show_confusion_matrix=True)
```

```
Accuracy on Test-Set: 30.0% (159 / 530)
Confusion Matrix:
[151   0   0] (0) forky
[122   3  12] (1) knifey
[237   0   5] (2) spoony
 (0) (1) (2)
```

1000次优化迭代后的性能

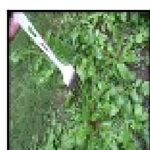
在1000次优化迭代之后，测试集上的准确率大约为70%。

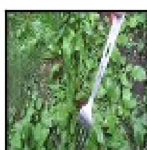
```
optimize(num_iterations=1000)
```

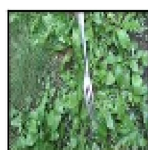
```
Global Step:    100, Training Batch Accuracy:  95.3%
Global Step:    200, Training Batch Accuracy:  96.9%
Global Step:    300, Training Batch Accuracy:  98.4%
Global Step:    400, Training Batch Accuracy: 100.0%
Global Step:    500, Training Batch Accuracy: 100.0%
Global Step:    600, Training Batch Accuracy: 100.0%
Global Step:    700, Training Batch Accuracy: 100.0%
Global Step:    800, Training Batch Accuracy: 100.0%
Global Step:    900, Training Batch Accuracy: 100.0%
Global Step:   1000, Training Batch Accuracy: 100.0%
Time usage: 0:00:02
```

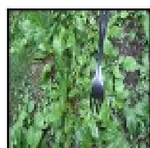
```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

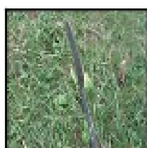
```
Accuracy on Test-Set: 71.3% (378 / 530)
Example errors:
```

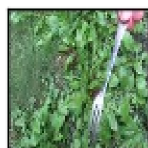

True: forky
Pred: spoony

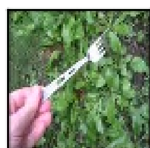

True: forky
Pred: spoony


True: forky
Pred: spoony

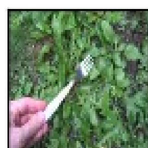

True: forky
Pred: spoony


True: knifey
Pred: spoony


True: forky
Pred: spoony


True: forky
Pred: knifey


True: knifey
Pred: forky


True: forky
Pred: knifey

```
Confusion Matrix:
[35 36 80] (0) forky
[  6 101  30] (1) knifey
[  0   0 242] (2) spoony
 (0) (1) (2)
```

关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。注意，我们需要关闭两个TensorFlow-session，每个模型对象各有一个。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# model.close()
# session.close()
```

总结

这篇教程向我们展示了如何在Inception模型上用自己的图像做迁移学习。教程中使用的几千张图像是我们用一个Python脚本从几分钟的录像视频中生成的。

然而，Knifey-Spoony数据集上的分类准确率不是很高，特别是叉子的图像。可能是因为Inception模型在ImageNet数据集上训练，而其中只有16张叉子的图像，但它却包括了1200多张勺子图像和1300多张餐刀图像。因此Inception模型很可能无法正确识别叉子。

因此我们需要另一种技巧来微调Inception模型，这样它就能更好地识别叉子。

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 试着为新的分类器改变神经网络。如果你删掉全连接层或添加更多的全连接层会发生什么？
- 如果你执行更多或更少的迭代会出现什么情况？
- 试着在训练集中删掉一些勺子的图像，这样每种类别的图象数量就差不多（先做个备份）。你还需要删除所有文件名带有 `*.pkl` 的缓存文件，然后重新运行Notebook。这样会提高分类准确率吗？比较改变前后的混淆矩阵。
- 用 `convert.py` 脚本建立你自己的数据集。比如，录下汽车和摩托车的视频，然后创建一个分类系统。

- 需要从你创建的训练集中删除一些不明确的图像吗？如何你删掉这些图像之后，分类准确率有什么变化？
- 改变Notebook，这样你可以输入单张图像而不是整个数据集。你不用从Inception模型中保存transfer-values。
- 你能创建一个比用Inception模型来做迁移学习更好的或更快的神经网络吗？
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #11

对抗样本

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube

中文翻译 thrillerist/Github

介绍

之前的教程中，我们用几种不用的深度神经网络来分类图像，取得不同程度的成功。在这篇教程里，我们将会看到一个寻找对抗样本的简单方法，它会使一个最先进的神经网络误分类任何输入图像，不管选的是什么类别。这通过简单地向输入图像添加小部分"特定"噪声完成。人类不会觉察到这些变化，但它却能戏弄神经网络。

本文基于之前的教程。你需要大概地熟悉神经网络（教程#01和#02），了解Inception模型（教程#07）也很有帮助。

流程图

我们使用教程#07中的Inception模型，然后修改/黑掉TensorFlow图，来寻找引起Inception模型误分类输入图像的对抗样本。

在下面的流程图中，我们在《查理和巧克力工厂》图像上添加了一些噪声，然后作为Inception模型的输入。最终目标是找到使Inception模型将图像误分类成我们目标类型的噪声，这边选择 `书柜` 类型（分类号300）。

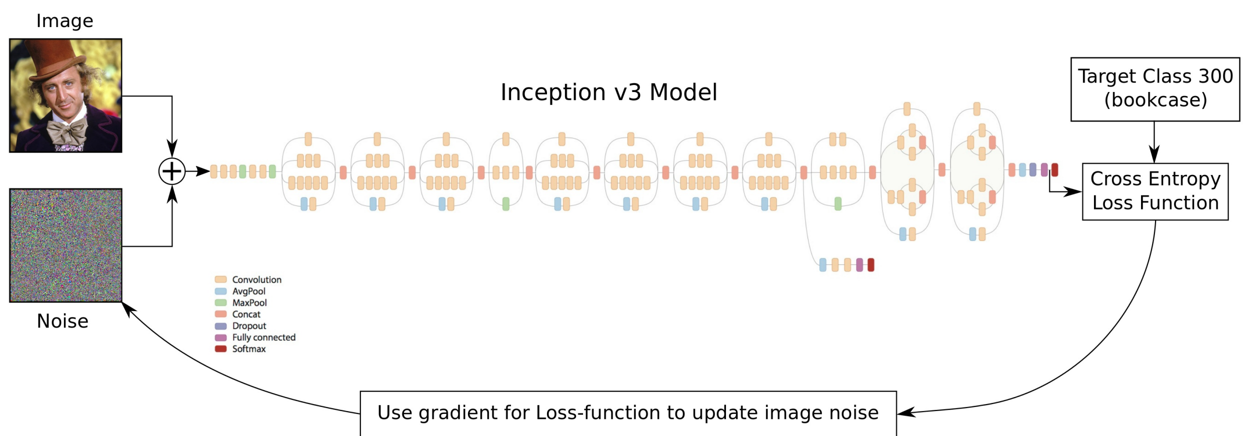
我们也为图添加一个新的损失函数，来计算cross-entropy，它是Inception模型分类噪声图像的性能度量。

由于Inception模型是由很多相结合的基本数学运算构造的，使用微分链式法则，TensorFlow让我们很快就能找到损失函数的梯度。

我们使用损失函数关于输入图像的梯度，来寻找对抗噪声。要寻找的是那些可以增加'书柜'类别而不是输入图像原始类别的评分（即概率）的噪声。

这本质上是用梯度下降法来执行优化的，后面会实现它。

```
from IPython.display import Image, display
Image('images/11_adversarial_examples_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import os

# Functions and classes for loading and using the Inception mode
l.
import inception
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.11.0rc0'
```

Inception 模型

从网上下载Inception模型。

从网上下载Inception模型。这是你保存数据文件的默认文件夹。如果文件夹不存在就自动创建。

```
# inception.data_dir = 'inception/'
```

如果文件夹中不存在Inception模型，就自动下载。 它有85MB。

```
inception.maybe_download()
```

```
Downloading Inception v3 Model ...
Data has apparently already been downloaded and unpacked.
```

载入Inception模型

载入模型，为图像分类做准备。

注意warning信息，以后可能会导致程序运行失败。

```
model = inception.Inception()
```

获取Inception模型的输入和输出

取得Inception模型输入张量的引用。这个张量是用来保存调整大小后的图像，即299 x 299像素并带有3个颜色通道。我们会在调整大小后的图像上添加噪声，然后还是用这个张量将结果传到图（graph）中，因此需要确保调整大小的算法没有引入噪声 。

```
resized_image = model.resized_image
```

获取Inception模型softmax分类器输出的引用。

```
y_pred = model.y_pred
```

获取Inception模型softmax分类器未经尺度变化的（unscaled）输出的引用。这通常称为“logits”。由于我们会在graph上添加一个新的损失函数，其中用到这些未经变化的输出，因此logits是必要的。

```
y_logits = model.y_logits
```

黑掉Inception模型

为了找到对抗样本，需要为Inception模型的图添加一个新的损失函数。我们还需要这个损失函数关于输入图像的梯度。

```
# Set the graph for the Inception model as the default graph,
# so that all changes inside this with-block are done to that gr
aph.
with model.graph.as_default():
    # Add a placeholder variable for the target class-number.
    # This will be set to e.g. 300 for the 'bookcase' class.
    pl_cls_target = tf.placeholder(dtype=tf.int32)

    # Add a new loss-function. This is the cross-entropy.
    # See Tutorial #01 for an explanation of cross-entropy.
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits
=y_logits, labels=[pl_cls_target])

    # Get the gradient for the loss-function with regard to
    # the resized input image.
    gradient = tf.gradients(loss, resized_image)
```

TensorFlow 会话

我们需要一个TensorFlow会话来运行图。

```
session = tf.Session(graph=model.graph)
```

帮助函数用来寻找对抗噪声

下面的函数找出了要添加到输入图像上的噪声，这样（输入图像）就会被分类到想要的目标类型。

这个函数本质上是用梯度下降来执行优化。噪声被初始化为零，然后用损失函数关于输入噪声图像的梯度来逐步优化，这样，每次迭代噪声都使分类更接近于想要的目标类型。当分类评分达到要求（比如99%）或者执行了最大迭代次数时，就停止优化。

```
def find_adversary_noise(image_path, cls_target, noise_limit=3.0
,
                         required_score=0.99, max_iterations=100)
:
    """
    Find the noise that must be added to the given image so
    that it is classified as the target-class.

    image_path: File-path to the input-image (must be *.jpg).
    cls_target: Target class-number (integer between 1-1000).
    noise_limit: Limit for pixel-values in the noise.
    required_score: Stop when target-class score reaches this.
    max_iterations: Max number of optimization iterations to per
```

```
form.
    """

    # Create a feed-dict with the image.
    feed_dict = model._create_feed_dict(image_path=image_path)

    # Use TensorFlow to calculate the predicted class-scores
    # (aka. probabilities) as well as the resized image.
    pred, image = session.run([y_pred, resized_image],
                              feed_dict=feed_dict)

    # Convert to one-dimensional array.
    pred = np.squeeze(pred)

    # Predicted class-number.
    cls_source = np.argmax(pred)

    # Score for the predicted class (aka. probability or confide
nce).
    score_source_org = pred.max()

    # Names for the source and target classes.
    name_source = model.name_lookup.cls_to_name(cls_source,
                                                only_first_name=
True)
    name_target = model.name_lookup.cls_to_name(cls_target,
                                                only_first_name=
True)

    # Initialize the noise to zero.
    noise = 0

    # Perform a number of optimization iterations to find
    # the noise that causes mis-classification of the input imag
e.
    for i in range(max_iterations):
        print("Iteration:", i)

        # The noisy image is just the sum of the input image and
 noise.
        noisy_image = image + noise

        # Ensure the pixel-values of the noisy image are between
        # 0 and 255 like a real image. If we allowed pixel-values

        # outside this range then maybe the mis-classification w
ould
        # be due to this 'illegal' input breaking the Inception
model.
        noisy_image = np.clip(a=noisy_image, a_min=0.0, a_max=25
5.0)

        # Create a feed-dict. This feeds the noisy image to the
```

```
        # tensor in the graph that holds the resized image, beca
use
        # this is the final stage for inputting raw image data.
        # This also feeds the target class-number that we desire.

        feed_dict = {model.tensor_name_resized_image: noisy_imag
e,
                     pl_cls_target: cls_target}

        # Calculate the predicted class-scores as well as the gr
adient.
        pred, grad = session.run([y_pred, gradient],
                                 feed_dict=feed_dict)

        # Convert the predicted class-scores to a one-dim array.
        pred = np.squeeze(pred)

        # The scores (probabilities) for the source and target c
lasses.
        score_source = pred[cls_source]
        score_target = pred[cls_target]

        # Squeeze the dimensionality for the gradient-array.
        grad = np.array(grad).squeeze()

        # The gradient now tells us how much we need to change t
he
        # noisy input image in order to move the predicted class
        # closer to the desired target-class.

        # Calculate the max of the absolute gradient values.
        # This is used to calculate the step-size.
        grad_absmax = np.abs(grad).max()

        # If the gradient is very small then use a lower limit,
        # because we will use it as a divisor.
        if grad_absmax < 1e-10:
            grad_absmax = 1e-10

        # Calculate the step-size for updating the image-noise.
        # This ensures that at least one pixel colour is changed
 by 7.
        # Recall that pixel colours can have 255 different value
s.
        # This step-size was found to give fast convergence.
        step_size = 7 / grad_absmax

        # Print the score etc. for the source-class.
        msg = "Source score: {0:>7.2%}, class-number: {1:>4}, cl
ass-name: {2}"
        print(msg.format(score_source, cls_source, name_source))

        # Print the score etc. for the target-class.
```

```
        msg = "Target score: {0:>7.2%}, class-number: {1:>4}, cl
ass-name: {2}"
        print(msg.format(score_target, cls_target, name_target))

        # Print statistics for the gradient.
        msg = "Gradient min: {0:>9.6f}, max: {1:>9.6f}, stepsize
: {2:>9.2f}"
        print(msg.format(grad.min(), grad.max(), step_size))

        # Newline.
        print()

        # If the score for the target-class is not high enough.
        if score_target < required_score:
            # Update the image-noise by subtracting the gradient
            # scaled by the step-size.
            noise -= step_size * grad

            # Ensure the noise is within the desired range.
            # This avoids distorting the image too much.
            noise = np.clip(a=noise,
                            a_min=-noise_limit,
                            a_max=noise_limit)
        else:
            # Abort the optimization because the score is high e
nough.
            break

    return image.squeeze(), noisy_image.squeeze(), noise, \
           name_source, name_target, \
           score_source, score_source_org, score_target
```

绘制图像和噪声的帮助函数

函数对输入做归一化，则输入值在0.0到1.0之间，这样才能正确的显示出噪声。

```
def normalize_image(x):
    # Get the min and max values for all pixels in the input.
    x_min = x.min()
    x_max = x.max()

    # Normalize so all values are between 0.0 and 1.0
    x_norm = (x - x_min) / (x_max - x_min)

    return x_norm
```

这个函数绘制了原始图像、噪声图像，以及噪声。它也显示了类别名和评分。

```
def plot_images(image, noise, noisy_image,
                name_source, name_target,
                score_source, score_source_org, score_target):
    """
    Plot the image, the noisy image and the noise.
    Also shows the class-names and scores.

    Note that the noise is amplified to use the full range of
    colours, otherwise if the noise is very low it would be
    hard to see.

    image: Original input image.
    noise: Noise that has been added to the image.
    noisy_image: Input image + noise.
    name_source: Name of the source-class.
    name_target: Name of the target-class.
    score_source: Score for the source-class.
    score_source_org: Original score for the source-class.
    score_target: Score for the target-class.
    """

    # Create figure with sub-plots.
    fig, axes = plt.subplots(1, 3, figsize=(10,10))

    # Adjust vertical spacing.
    fig.subplots_adjust(hspace=0.1, wspace=0.1)

    # Use interpolation to smooth pixels?
    smooth = True

    # Interpolation type.
    if smooth:
        interpolation = 'spline16'
    else:
        interpolation = 'nearest'

    # Plot the original image.
    # Note that the pixel-values are normalized to the [0.0, 1.0]

    # range by dividing with 255.
    ax = axes.flat[0]
    ax.imshow(image / 255.0, interpolation=interpolation)
    msg = "Original Image:\n{0} ({1:.2%})"
    xlabel = msg.format(name_source, score_source_org)
    ax.set_xlabel(xlabel)

    # Plot the noisy image.
    ax = axes.flat[1]
    ax.imshow(noisy_image / 255.0, interpolation=interpolation)
    msg = "Image + Noise:\n{0} ({1:.2%})\n{2} ({3:.2%})"
    xlabel = msg.format(name_source, score_source, name_target,
score_target)
```

```
    ax.set_xlabel(xlabel)

    # Plot the noise.
    # The colours are amplified otherwise they would be hard to
see.
    ax = axes.flat[2]
    ax.imshow(normalize_image(noise), interpolation=interpolatio
n)
    xlabel = "Amplified Noise"
    ax.set_xlabel(xlabel)

    # Remove ticks from all the plots.
    for ax in axes.flat:
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

寻找并绘制对抗样本的帮助函数

这个函数结合了上面的两个方法。它先找到对抗噪声，然后画出图像和噪声。

```
def adversary_example(image_path, cls_target,
                      noise_limit, required_score):
    """
    Find and plot adversarial noise for the given image.

    image_path: File-path to the input-image (must be *.jpg).
    cls_target: Target class-number (integer between 1-1000).
    noise_limit: Limit for pixel-values in the noise.
    required_score: Stop when target-class score reaches this.
    """

    # Find the adversarial noise.
    image, noisy_image, noise, \
    name_source, name_target, \
    score_source, score_source_org, score_target = \
        find_adversary_noise(image_path=image_path,
                             cls_target=cls_target,
                             noise_limit=noise_limit,
                             required_score=required_score)

    # Plot the image and the noise.
    plot_images(image=image, noise=noise, noisy_image=noisy_imag
e,
                name_source=name_source, name_target=name_target
,
                score_source=score_source,
                score_source_org=score_source_org,
                score_target=score_target)

    # Print some statistics for the noise.
    msg = "Noise min: {0:.3f}, max: {1:.3f}, mean: {2:.3f}, std:
 {3:.3f}"
    print(msg.format(noise.min(), noise.max(),
                     noise.mean(), noise.std()))
```

结果

鹦鹉

这个例子将一张鹦鹉图作为输入，然后找到对抗噪声，使得Inception模型将图像误分类成一个书架（类别号300）。

噪声界限设为3.0，这表示只允许每个像素颜色在3.0范围内波动。像素颜色在0到255之间，因此3.0的浮动对应大约1.2%的可能范围。这样的少量噪声对人眼是不可见的，因此噪声图像和原始图像看起来基本一致，如下所示。

要求评分设为0.99，这表示当目标分类的评分大于等于0.99时，用来寻找对抗噪声的优化器就会停止，这样Inception模型几乎确定了噪声图像展示的是期望的目标类别。

```
image_path = "images/parrot_cropped1.jpg"

adversary_example(image_path=image_path,
                  cls_target=300,
                  noise_limit=3.0,
                  required_score=0.99)
```

```
Iteration: 0
Source score:  97.38%, class-number:  409, class-name: macaw
Target score:   0.00%, class-number:  300, class-name: bookcase
Gradient min: -0.001329, max:  0.001370, stepsize:   5110.94

Iteration: 1
Source score:  88.87%, class-number:  409, class-name: macaw
Target score:   0.01%, class-number:  300, class-name: bookcase
Gradient min: -0.001499, max:  0.001401, stepsize:   4668.28

Iteration: 2
Source score:  68.47%, class-number:  409, class-name: macaw
Target score:   0.06%, class-number:  300, class-name: bookcase
Gradient min: -0.003093, max:  0.002587, stepsize:   2262.91

Iteration: 3
Source score:  16.76%, class-number:  409, class-name: macaw
Target score:   0.22%, class-number:  300, class-name: bookcase
Gradient min: -0.001077, max:  0.001047, stepsize:   6499.39

Iteration: 4
Source score:  31.76%, class-number:  409, class-name: macaw
Target score:   0.41%, class-number:  300, class-name: bookcase
Gradient min: -0.001670, max:  0.001715, stepsize:   4081.82

Iteration: 5
Source score:  11.86%, class-number:  409, class-name: macaw
Target score:   0.72%, class-number:  300, class-name: bookcase
Gradient min: -0.001524, max:  0.002019, stepsize:   3466.85

Iteration: 6
Source score:   2.41%, class-number:  409, class-name: macaw
Target score:   3.26%, class-number:  300, class-name: bookcase
Gradient min: -0.001685, max:  0.001247, stepsize:   4154.00

Iteration: 7
Source score:   3.02%, class-number:  409, class-name: macaw
Target score:   7.07%, class-number:  300, class-name: bookcase
Gradient min: -0.001503, max:  0.001707, stepsize:   4101.29

Iteration: 8
Source score:   2.34%, class-number:  409, class-name: macaw
Target score:   6.59%, class-number:  300, class-name: bookcase
```

```
Gradient min: -0.003677, max:  0.003430, stepsize:   1903.80

Iteration: 9
Source score:   1.33%, class-number:  409, class-name: macaw
Target score:  16.10%, class-number:  300, class-name: bookcase
Gradient min: -0.001366, max:  0.001558, stepsize:   4492.61

Iteration: 10
Source score:   0.85%, class-number:  409, class-name: macaw
Target score:  14.19%, class-number:  300, class-name: bookcase
Gradient min: -0.001632, max:  0.001372, stepsize:   4288.61

Iteration: 11
Source score:   0.89%, class-number:  409, class-name: macaw
Target score:  38.05%, class-number:  300, class-name: bookcase
Gradient min: -0.001264, max:  0.000991, stepsize:   5539.81

Iteration: 12
Source score:   0.44%, class-number:  409, class-name: macaw
Target score:  35.43%, class-number:  300, class-name: bookcase
Gradient min: -0.001744, max:  0.002125, stepsize:   3293.86

Iteration: 13
Source score:   0.29%, class-number:  409, class-name: macaw
Target score:  60.42%, class-number:  300, class-name: bookcase
Gradient min: -0.000611, max:  0.000705, stepsize:   9927.19

Iteration: 14
Source score:   0.24%, class-number:  409, class-name: macaw
Target score:  40.47%, class-number:  300, class-name: bookcase
Gradient min: -0.001014, max:  0.001096, stepsize:   6385.38

Iteration: 15
Source score:   1.98%, class-number:  409, class-name: macaw
Target score:  41.95%, class-number:  300, class-name: bookcase
Gradient min: -0.001578, max:  0.001865, stepsize:   3753.93

Iteration: 16
Source score:   0.04%, class-number:  409, class-name: macaw
Target score:  78.76%, class-number:  300, class-name: bookcase
Gradient min: -0.000333, max:  0.000335, stepsize:  20888.12

Iteration: 17
Source score:   1.93%, class-number:  409, class-name: macaw
Target score:  43.73%, class-number:  300, class-name: bookcase
Gradient min: -0.001840, max:  0.002724, stepsize:   2569.94

Iteration: 18
Source score:   0.02%, class-number:  409, class-name: macaw
Target score:  91.74%, class-number:  300, class-name: bookcase
Gradient min: -0.000328, max:  0.000189, stepsize:  21342.00

Iteration: 19
```

```
Source score:   0.00%, class-number:  409, class-name: macaw
Target score:  97.37%, class-number:  300, class-name: bookcase
Gradient min: -0.000064, max:  0.000084, stepsize:  83366.77

Iteration: 20
Source score:   0.01%, class-number:  409, class-name: macaw
Target score:  97.13%, class-number:  300, class-name: bookcase
Gradient min: -0.000089, max:  0.000086, stepsize:  78565.60

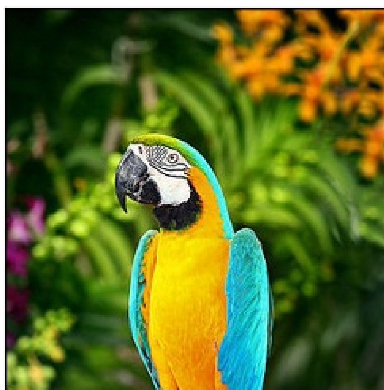
Iteration: 21
Source score:   0.01%, class-number:  409, class-name: macaw
Target score:  94.92%, class-number:  300, class-name: bookcase
Gradient min: -0.000128, max:  0.000142, stepsize:  49304.41

Iteration: 22
Source score:   0.01%, class-number:  409, class-name: macaw
Target score:  97.18%, class-number:  300, class-name: bookcase
Gradient min: -0.000071, max:  0.000058, stepsize:  97917.04

Iteration: 23
Source score:   0.01%, class-number:  409, class-name: macaw
Target score:  95.90%, class-number:  300, class-name: bookcase
Gradient min: -0.000111, max:  0.000142, stepsize:  49346.70

Iteration: 24
Source score:   0.00%, class-number:  409, class-name: macaw
Target score:  98.98%, class-number:  300, class-name: bookcase
Gradient min: -0.000029, max:  0.000025, stepsize: 245266.90

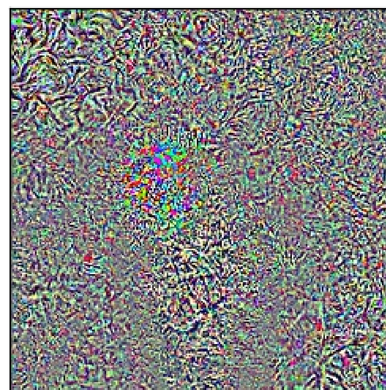
Iteration: 25
Source score:   0.00%, class-number:  409, class-name: macaw
Target score:  99.12%, class-number:  300, class-name: bookcase
Gradient min: -0.000019, max:  0.000022, stepsize: 311258.06
```



Original Image:
macaw (97.38%)



Image + Noise:
macaw (0.00%)
bookcase (99.12%)



Amplified Noise

```
Noise min: -3.000, max: 3.000, mean: 0.001, std: 1.492
```

如上所示，鹦鹉的原始图像与噪声图像看起来几乎一致。人眼无法区分开两张图像。原始图被Inception模型正确地分类成金刚鹦鹉（鹦鹉），评分为97.38%。但噪声图像对金刚鹦鹉的分类评分是0.00%，对书架的评分是99.12%。

这样，我们糊弄了Inception模型，让它相信一张鹦鹉图像展示的是一个书架。只是添加了一些“特定的”噪声就导致了这个误分类。

注意，上面展示的噪声是被放大数倍的。实际上，噪声只在输入图像每个像素颜色强度的最多1.2%范围内调整图像（假定噪声界限像上面的函数一样设置为3.0）。由于噪声很弱，人类观察不到，但它导致Inception模型完全误分类的输入图像。

Elon Musk

我们也找到了Elon Mask图像的对抗噪声。目标类别再次设为“书柜”（类别号300），噪声界限和要求分数也与上面的相同。

```
image_path = "images/elon_musk.jpg"

adversary_example(image_path=image_path,
                  cls_target=300,
                  noise_limit=3.0,
                  required_score=0.99)
```

```
Iteration: 0
Source score:  19.73%, class-number:  837, class-name: sweatshir
t
Target score:   0.01%, class-number:  300, class-name: bookcase
Gradient min: -0.008348, max:  0.005946, stepsize:    838.48

Iteration: 1
Source score:   1.77%, class-number:  837, class-name: sweatshir
t
Target score:   0.24%, class-number:  300, class-name: bookcase
Gradient min: -0.002952, max:  0.005907, stepsize:   1185.13

Iteration: 2
Source score:   0.52%, class-number:  837, class-name: sweatshir
t
Target score:  10.06%, class-number:  300, class-name: bookcase
Gradient min: -0.006741, max:  0.006555, stepsize:   1038.46

Iteration: 3
Source score:   0.24%, class-number:  837, class-name: sweatshir
t
Target score:  67.35%, class-number:  300, class-name: bookcase
Gradient min: -0.001548, max:  0.001130, stepsize:   4521.39

Iteration: 4
Source score:   0.01%, class-number:  837, class-name: sweatshir
t
Target score:  68.76%, class-number:  300, class-name: bookcase
Gradient min: -0.001654, max:  0.001889, stepsize:   3706.45

Iteration: 5
Source score:   0.12%, class-number:  837, class-name: sweatshir
t
Target score:  84.91%, class-number:  300, class-name: bookcase
Gradient min: -0.001288, max:  0.001800, stepsize:   3889.91

Iteration: 6
Source score:   0.00%, class-number:  837, class-name: sweatshir
t
Target score:  99.09%, class-number:  300, class-name: bookcase
Gradient min: -0.000029, max:  0.000021, stepsize: 244856.71
```


Original Image:
sweatshirt (19.73%)


Image + Noise:
sweatshirt (0.00%)
bookcase (99.09%)


Amplified Noise

```
Noise min: -3.000, max: 3.000, mean: -0.001, std: 0.668
```

Inception模型弄不太清原始输入图像的分类，认为它有可能是一件运动衫（评分19.73%）。但我们还是能够生成一个使Inception模型完全认为噪声图像是书架（评分99.09%）的对抗噪声，即使在人眼看来，两张图像几乎一样。

查理和巧克力工厂 (新版)

```
image_path = "images/willy_wonka_new.jpg"

adversary_example(image_path=image_path,
                  cls_target=300,
                  noise_limit=3.0,
                  required_score=0.99)
```

```
Iteration: 0
Source score:  31.48%, class-number:  535, class-name: sunglasse
s
Target score:   0.03%, class-number:  300, class-name: bookcase
Gradient min: -0.002181, max:  0.001478, stepsize:   3210.13

Iteration: 1
Source score:   2.08%, class-number:  535, class-name: sunglasse
s
Target score:   0.13%, class-number:  300, class-name: bookcase
Gradient min: -0.001447, max:  0.001573, stepsize:   4449.85

Iteration: 2
Source score:   6.37%, class-number:  535, class-name: sunglasse
s
Target score:   0.35%, class-number:  300, class-name: bookcase
Gradient min: -0.001421, max:  0.001633, stepsize:   4286.13
```

```
Iteration: 3
Source score:   2.25%, class-number:  535, class-name: sunglasse
s
Target score:   1.03%, class-number:  300, class-name: bookcase
Gradient min: -0.001736, max:  0.001874, stepsize:   3734.86

Iteration: 4
Source score:  10.54%, class-number:  535, class-name: sunglasse
s
Target score:   1.32%, class-number:  300, class-name: bookcase
Gradient min: -0.002901, max:  0.002503, stepsize:   2413.04

Iteration: 5
Source score:   1.86%, class-number:  535, class-name: sunglasse
s
Target score:   3.22%, class-number:  300, class-name: bookcase
Gradient min: -0.001784, max:  0.001904, stepsize:   3675.68

Iteration: 6
Source score:   2.19%, class-number:  535, class-name: sunglasse
s
Target score:   5.44%, class-number:  300, class-name: bookcase
Gradient min: -0.002405, max:  0.001714, stepsize:   2911.17

Iteration: 7
Source score:   4.16%, class-number:  535, class-name: sunglasse
s
Target score:   3.61%, class-number:  300, class-name: bookcase
Gradient min: -0.001463, max:  0.002057, stepsize:   3402.83

Iteration: 8
Source score:   2.25%, class-number:  535, class-name: sunglasse
s
Target score:  19.46%, class-number:  300, class-name: bookcase
Gradient min: -0.003193, max:  0.001512, stepsize:   2192.48

Iteration: 9
Source score:   1.25%, class-number:  535, class-name: sunglasse
s
Target score:  50.62%, class-number:  300, class-name: bookcase
Gradient min: -0.000910, max:  0.000770, stepsize:   7693.95

Iteration: 10
Source score:   0.86%, class-number:  535, class-name: sunglasse
s
Target score:  37.99%, class-number:  300, class-name: bookcase
Gradient min: -0.001351, max:  0.001484, stepsize:   4718.11

Iteration: 11
Source score:   6.40%, class-number:  535, class-name: sunglasse
s
Target score:  27.42%, class-number:  300, class-name: bookcase
Gradient min: -0.001785, max:  0.001544, stepsize:   3920.83
```

```
Iteration: 12
Source score:   0.17%, class-number:  535, class-name: sunglasse
s
Target score:  73.86%, class-number:  300, class-name: bookcase
Gradient min: -0.000646, max:  0.000842, stepsize:   8315.79

Iteration: 13
Source score:   0.16%, class-number:  535, class-name: sunglasse
s
Target score:  89.56%, class-number:  300, class-name: bookcase
Gradient min: -0.000217, max:  0.000296, stepsize:  23618.89

Iteration: 14
Source score:   0.19%, class-number:  535, class-name: sunglasse
s
Target score:  89.90%, class-number:  300, class-name: bookcase
Gradient min: -0.000196, max:  0.000241, stepsize:  29075.62

Iteration: 15
Source score:   0.28%, class-number:  535, class-name: sunglasse
s
Target score:  87.20%, class-number:  300, class-name: bookcase
Gradient min: -0.000232, max:  0.000209, stepsize:  30222.49

Iteration: 16
Source score:   0.99%, class-number:  535, class-name: sunglasse
s
Target score:  75.64%, class-number:  300, class-name: bookcase
Gradient min: -0.000799, max:  0.000592, stepsize:   8761.73

Iteration: 17
Source score:   0.06%, class-number:  535, class-name: sunglasse
s
Target score:  96.55%, class-number:  300, class-name: bookcase
Gradient min: -0.000078, max:  0.000057, stepsize:  90126.50

Iteration: 18
Source score:   0.26%, class-number:  535, class-name: sunglasse
s
Target score:  85.38%, class-number:  300, class-name: bookcase
Gradient min: -0.000487, max:  0.000490, stepsize:  14284.58

Iteration: 19
Source score:   0.25%, class-number:  535, class-name: sunglasse
s
Target score:  93.26%, class-number:  300, class-name: bookcase
Gradient min: -0.000143, max:  0.000156, stepsize:  44844.46

Iteration: 20
Source score:   0.07%, class-number:  535, class-name: sunglasse
s
Target score:  93.84%, class-number:  300, class-name: bookcase
```

```
Gradient min: -0.000166, max:  0.000141, stepsize:  42205.53

Iteration: 21
Source score:   0.03%, class-number:  535, class-name: sunglasse
s
Target score:  98.31%, class-number:  300, class-name: bookcase
Gradient min: -0.000033, max:  0.000026, stepsize: 213124.72

Iteration: 22
Source score:   0.03%, class-number:  535, class-name: sunglasse
s
Target score:  98.80%, class-number:  300, class-name: bookcase
Gradient min: -0.000023, max:  0.000027, stepsize: 260036.19

Iteration: 23
Source score:   0.03%, class-number:  535, class-name: sunglasse
s
Target score:  99.03%, class-number:  300, class-name: bookcase
Gradient min: -0.000022, max:  0.000024, stepsize: 294094.62
```



Original Image:
sunglasses (31.48%)



Image + Noise:
sunglasses (0.03%)
bookcase (99.03%)



Amplified Noise

```
Noise min: -3.000, max: 3.000, mean: 0.010, std: 1.534
```

在上面的《查理和巧克力工厂》图像中（新版电影），原先Inception模型将图像分类成“太阳镜”（评分31.48%）。但再一次，我们能够生成让模型将图像分类成“书架”的对抗噪声（评分99.03%）。

两张图像看起来一样。但你可以倾斜电脑屏幕，看到白色区域一些轻微变化的噪声图样。

查理和巧克力工厂 (旧版)

```
image_path = "images/willy_wonka_old.jpg"

adversary_example(image_path=image_path,
                  cls_target=300,
                  noise_limit=3.0,
                  required_score=0.99)
```

```
Iteration: 0
Source score:  97.22%, class-number:  817, class-name: bow tie
Target score:   0.00%, class-number:  300, class-name: bookcase
Gradient min: -0.002479, max:  0.003469, stepsize:   2017.94

Iteration: 1
Source score:  10.65%, class-number:  817, class-name: bow tie
Target score:   0.08%, class-number:  300, class-name: bookcase
Gradient min: -0.000859, max:  0.001458, stepsize:   4799.50

Iteration: 2
Source score:   2.21%, class-number:  817, class-name: bow tie
Target score:   0.25%, class-number:  300, class-name: bookcase
Gradient min: -0.000415, max:  0.000617, stepsize:  11350.70

Iteration: 3
Source score:   3.59%, class-number:  817, class-name: bow tie
Target score:   0.74%, class-number:  300, class-name: bookcase
Gradient min: -0.000643, max:  0.000752, stepsize:   9304.24

Iteration: 4
Source score:   3.05%, class-number:  817, class-name: bow tie
Target score:   1.42%, class-number:  300, class-name: bookcase
Gradient min: -0.000744, max:  0.000688, stepsize:   9407.59

Iteration: 5
Source score:   1.80%, class-number:  817, class-name: bow tie
Target score:   1.35%, class-number:  300, class-name: bookcase
Gradient min: -0.000924, max:  0.000954, stepsize:   7334.48

Iteration: 6
Source score:   9.09%, class-number:  817, class-name: bow tie
Target score:   3.70%, class-number:  300, class-name: bookcase
Gradient min: -0.002771, max:  0.003224, stepsize:   2171.03

Iteration: 7
Source score:   1.05%, class-number:  817, class-name: bow tie
Target score:  15.34%, class-number:  300, class-name: bookcase
Gradient min: -0.001409, max:  0.001925, stepsize:   3637.15

Iteration: 8
Source score:   1.58%, class-number:  817, class-name: bow tie
Target score:  32.90%, class-number:  300, class-name: bookcase
```

```
Gradient min: -0.001282, max:  0.001393, stepsize:   5023.51

Iteration: 9
Source score:   0.98%, class-number:  817, class-name: bow tie
Target score:  32.66%, class-number:  300, class-name: bookcase
Gradient min: -0.001728, max:  0.001736, stepsize:   4032.38

Iteration: 10
Source score:   0.59%, class-number:  817, class-name: bow tie
Target score:  66.56%, class-number:  300, class-name: bookcase
Gradient min: -0.000976, max:  0.000736, stepsize:   7173.06

Iteration: 11
Source score:   0.10%, class-number:  817, class-name: bow tie
Target score:  85.64%, class-number:  300, class-name: bookcase
Gradient min: -0.000260, max:  0.000254, stepsize:  26939.47

Iteration: 12
Source score:   0.15%, class-number:  817, class-name: bow tie
Target score:  89.87%, class-number:  300, class-name: bookcase
Gradient min: -0.000341, max:  0.000252, stepsize:  20529.36

Iteration: 13
Source score:   0.00%, class-number:  817, class-name: bow tie
Target score:  98.09%, class-number:  300, class-name: bookcase
Gradient min: -0.000037, max:  0.000041, stepsize: 168840.03

Iteration: 14
Source score:   0.07%, class-number:  817, class-name: bow tie
Target score:  95.18%, class-number:  300, class-name: bookcase
Gradient min: -0.000212, max:  0.000168, stepsize:  32997.19

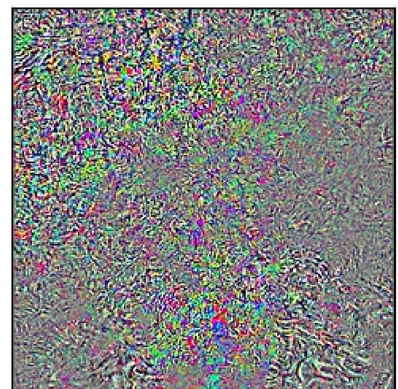
Iteration: 15
Source score:   0.00%, class-number:  817, class-name: bow tie
Target score:  99.72%, class-number:  300, class-name: bookcase
Gradient min: -0.000004, max:  0.000004, stepsize: 1590352.60
```



Original Image:
bow tie (97.22%)



Image + Noise:
bow tie (0.00%)
bookcase (99.72%)



Amplified Noise

```
Noise min: -3.000, max: 3.000, mean: -0.000, std: 1.309
```

《查理和巧克力工厂》图像（旧版电影）原先被Inception模型分类成“蝴蝶领结”。同样，加了噪声之后，它被分类成“书架”（评分99.72%）。

关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。注意，我们需要关闭两个TensorFlow-session，每个模型对象各有一个。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# session.close()
# model.close()
```

总结

我们演示了如何寻找导致Inception模型误分类图像的对抗样本。通过一个简单的流程，我们发现将噪声添加到输入图像上会使模型错误地分类图像，即使每个像素只做了轻微的改变，而且人眼无法察觉这些变化。

更进一步，优化后的噪声可以给出一个接近100%的评分（概率或确信度）。因此，输入图像不仅被误分类了，神经网络还很确信自己正确地分类了图像。

这是神经网络的一个普遍的问题，并且是一个很严肃的问题！我们无法在关键应用中相信神经网络，直到能够理解为什么会发生上述问题或如何解决它。想象一下自动驾驶汽车由于其神经网络误分类了输入图像而忽视停止标志或穿过马路的行人。

对这个问题的研究正在进行中，鼓励你在网上搜索一下这个课题的最新论文。也许你可以找到问题的解决方案？

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 试着使用自己的图像。
- 试着在 `adversary_example()` 中使用其他的参数。试试其它的目标类别、噪声界限和评分要求。结果是怎样的？
- 你认为对于所有的目标类别都能生成它的对抗噪声吗？如何证明你的理论？
- 试着在 `find_adversary_noise()` 中使用不同的公式来计算step-size。你能

使优化更快吗？
- 试着在噪声图像输入到神经网络之前对它进行模糊处理。它能去掉对抗噪声，并且导致再一次的正确分类吗？
- 试着降低噪声图像的颜色深度，而不是对它做模糊。它会去除对抗噪声并导致正确分类吗？比如将图像的RGB限制在16或32位里，通常是有255位的。
- 你认为你的噪声消除对MNIST数据集的手写数字或奇特的几何形状有效吗？有时将这些称为'fooling images'，上网搜索看看。
- 你能找到对所有图像都有效的对抗噪声吗？这样就不用为每张图像寻找特定的噪声了。你会怎么做？
- 你能直接用TensorFlow而不是Numpy来实现 `find_adversary_noise()` 吗？需要在TensorFlow图中创建一个噪声变量，这样它就能被优化。
- 向朋友解释什么是对抗样本以及程序如何找到它们。

License (MIT)

TensorFlow 教程 #12

MNIST的对抗噪声

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

介绍

之前的教程#11展示了如何找到最先进神经网络的对抗样本，它会引起网络误分类图像，即使在人眼看来图像完全相同。例如，在添加了对抗噪声之后，一张鹦鹉的图像会被误分类成书架，但在人类眼中图像完全没什么变化。

教程#11是通过每张图像的优化过程来寻找对抗噪声的。由于噪声是专门为某张图像生成，因此它可能不是通用的，无法在其他图像上起作用。

本教程将会找到那些导致几乎所有输入图像都被误分类成目标类别的对抗噪声。我们使用MNIST手写数字数据集为例。现在，对抗噪声对人眼是清晰可见的，但人类还是能够很容易地辨认出数字，然而神经网络几乎将所有图像误分类。

这篇教程里，我们还会试着让神经网络对对抗噪声免疫。

教程 #11 用Numpy来做对抗优化。在这篇教程里，我们会直接在TensorFlow里实现优化过程。这会更快速，尤其是在使用GPU的时候，因为不用每次迭代都在GPU里拷贝数据。

推荐你先学习教程 #11。你也需要大概地熟悉神经网络，详见教程 #01和 #02。

流程图

下面的图表直接展示了之后实现的卷积神经网络中数据的传递。

例子展示的是数字7的输入图像。随后在图像上添加对抗噪声。红色的噪声点是正值，它让像素值更深，蓝色噪声点是负值，让输入图像在此处的颜色更浅。

这些噪声图像传到神经网络中，然后得到一个预测数字。这种情况下，对抗噪声让神经网络相信这张数字7的图像显示的是数字3。噪声对人眼是清晰可见的，但人类仍然可以容易地辨认出数字7来。

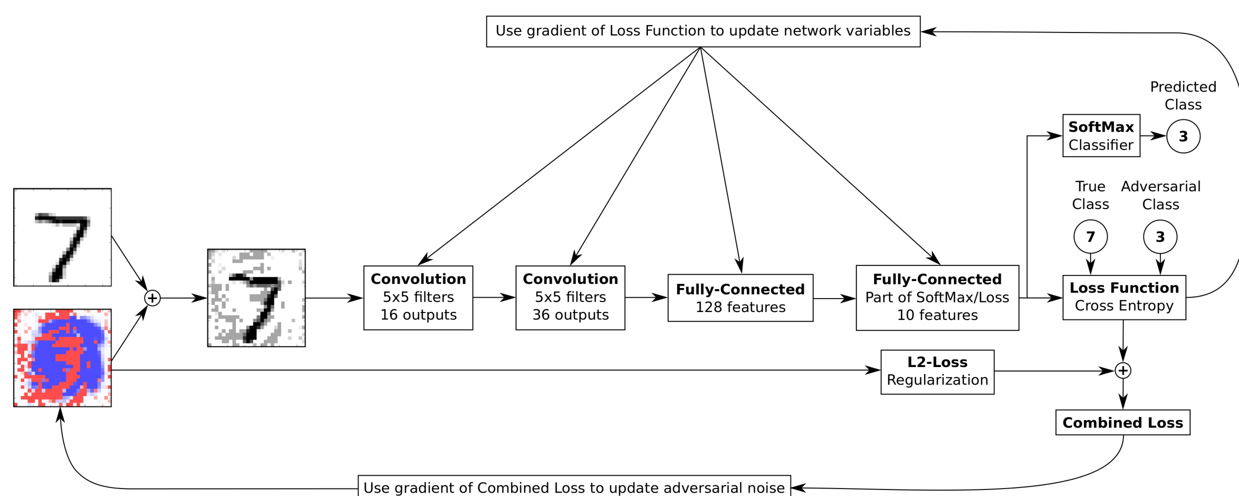
这边值得注意的是，单一的噪声模式会导致神经网络将几乎所有的输入图像都误分类成期望的目标类型。

在这个神经网络中有两个单独的优化程序。首先，我们优化神经网络的变量来分类训练集的图像。这是神经网络的常规优化过程。一旦分类准确率足够高，我们就切换到第二个优化程序，（它用来）寻找单一模式的对抗噪声，使得所有的输入图像

都被误分类成目标类型。

这两个优化程序是完全独立的。第一个程序只修改量神经网络的变量，第二个程序只修改对抗噪声。

```
from IPython.display import Image
Image('images/12_adversarial_noise_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
import time
from datetime import timedelta
import math

# We also need PrettyTensor.
import prettytensor as pt
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.12.0-rc0'
```

PrettyTensor 版本:

```
pt.__version__
```

```
'0.7.1'
```

载入数据

MNIST数据集大约12MB，如果没在给定路径中找到就会自动下载。

```
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets('data/MNIST/', one_hot=True)
```

```
Extracting data/MNIST/train-images-idx3-ubyte.gz
Extracting data/MNIST/train-labels-idx1-ubyte.gz
Extracting data/MNIST/t10k-images-idx3-ubyte.gz
Extracting data/MNIST/t10k-labels-idx1-ubyte.gz
```

现在已经载入了MNIST数据集，它由70,000张图像和对应的标签（比如图像的类别）组成。数据集分成三份互相独立的子集。我们在教程中只用训练集和测试集。

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(data.train.labels)))
print("- Test-set:\t\t{}".format(len(data.test.labels)))
print("- Validation-set:\t{}".format(len(data.validation.labels)
))
```

```
Size of:
- Training-set:		55000
- Test-set:		10000
- Validation-set:	5000
```

类型标签使用One-Hot编码，这意外每个标签是长为10的向量，除了一个元素之外，其他的都为零。这个元素的索引就是类别的数字，即相应图片中画的数字。我们也需要测试数据集类别数字的整型值，现在计算它。

```
data.test.cls = np.argmax(data.test.labels, axis=1)
```

数据维度

在下面的源码中，有很多地方用到了数据维度。它们只在一个地方定义，因此我们可以在代码中使用这些数字而不是直接写数字。

```
# We know that MNIST images are 28 pixels in each dimension.
img_size = 28

# Images are stored in one-dimensional arrays of this length.
img_size_flat = img_size * img_size

# Tuple with height and width of images used to reshape arrays.
img_shape = (img_size, img_size)

# Number of colour channels for the images: 1 channel for gray-s
cale.
num_channels = 1

# Number of classes, one class for each of 10 digits.
num_classes = 10
```

用来绘制图像的帮助函数

这个函数用来在3x3的栅格中画9张图像，然后在每张图像下面写出真实类别和预测类别。如果提供了噪声，就将其添加到所有图像上。

```
def plot_images(images, cls_true, cls_pred=None, noise=0.0):
    assert len(images) == len(cls_true) == 9

    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Get the i'th image and reshape the array.
        image = images[i].reshape(img_shape)

        # Add the adversarial noise to the image.
        image += noise

        # Ensure the noisy pixel-values are between 0 and 1.
        image = np.clip(image, 0.0, 1.0)

        # Plot image.
        ax.imshow(image,
                  cmap='binary', interpolation='nearest')

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i],
cls_pred[i])

        # Show the classes as the label on the x-axis.
        ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

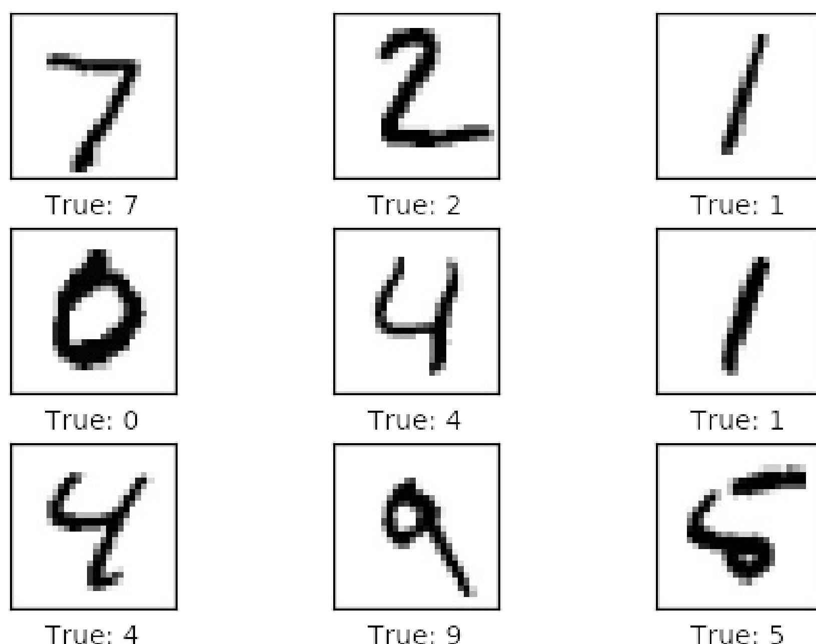
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

绘制几张图像来看看数据是否正确

```
# Get the first images from the test-set.
images = data.test.images[0:9]

# Get the true classes for those images.
cls_true = data.test.cls[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true)
```



TensorFlow图（Graph）

现在将使用TensorFlow和PrettyTensor构建神经网络的计算图。 与往常一样，我们需要为图像创建占位符变量，将其送到计算图中，然后将对抗噪声添加到图像中。接着把噪声图像用作卷积神经网络的输入。

这个网络有两个单独的优化程序。神经网络本身变量的一个常规优化过程，以及对抗噪声的另一个优化过程。 两个优化过程都直接在TensorFlow中实现。

占位符（Placeholder）变量

占位符变量为TensorFlow中的计算图提供了输入，我们可以在每次执行图的时候更改。 我们称为feeding占位符变量。

首先，我们为输入图像定义占位符变量。 这允许我们改变输入到TensorFlow图中的图像。 这是一个张量，代表它是一个多维数组。 数据类型设为 `float32` ，形状设为 `[None，img_size_flat]` ，其中 `None` 代表张量可以保存任意数量的图像，每个图像是长度为 `img_size_flat` 的向量。

```
x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name=
'x')
```

卷积层希望 `x` 被编码为4维张量，因此我们需要将它的形状转换至 `[num_images, img_height, img_width, num_channels]` 。注意 `img_height == img_width == img_size` ，如果第一维的大小设为-1，num_images的大小也会被自动推导出来。转换运算如下：

```
x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
```

接下来我们为输入变量x中的图像所对应的真实标签定义占位符变量。变量的形状是 `[None, num_classes]` ，这代表着它保存了任意数量的标签，每个标签是长度为 `num_classes` 的向量，本例中长度为10。

```
y_true = tf.placeholder(tf.float32, shape=[None, num_classes], name='y_true')
```

我们也可以为类别号提供一个占位符，但这里用argmax来计算它。这里只是TensorFlow中的一些操作符，没有执行什么运算。

```
y_true_cls = tf.argmax(y_true, dimension=1)
```

对抗噪声

输入图像的像素值在0.0到1.0之间。对抗噪声是在输入图像上添加或删除的数值。

对抗噪声的界限设为0.35，则噪声在正负0.35之间。

```
noise_limit = 0.35
```

对抗噪声的优化器会试图最小化两个损失度量：(1)神经网络常规的损失度量，因此我们会找到使得目标类型分类准确率最高的噪声；(2)L2-loss度量，它会保持尽可能低的噪声。

下面的权重决定了与常规的损失度量相比，L2-loss的重要性。通常接近零的L2权重表现的更好。

```
noise_l2_weight = 0.02
```

当我们为噪声创建变量时，必须告知TensorFlow它属于哪一个变量集合，这样，后面就能通知两个优化器要更新哪些变量。

首先为变量集合定义一个名称。这只是一个字符串。

```
ADVERSARY_VARIABLES = 'adversary_variables'
```

接着，创建噪声变量所属集合的列表。如果我们将噪声变量添加到集合 `tf.GraphKeys.VARIABLES` 中，它就会和TensorFlow图中的其他变量一起被初始化，但不会被优化。这里有点混乱。

```
collections = [tf.GraphKeys.VARIABLES, ADVERSARY_VARIABLES]
```

现在我们可以为对抗噪声添加新的变量。它会被初始化为零。它是不可训练的，因此并不会与神经网络中的其他变量一起被优化。这让我们可以创建两个独立的优化程序。

```
x_noise = tf.Variable(tf.zeros([img_size, img_size, num_channels
]),
                      name='x_noise', trainable=False,
                      collections=collections)
```

对抗噪声会被限制在我们上面设定的噪声界限内。注意此时并未在计算图表内进行计算，在优化步骤之后执行，详见下文。

```
x_noise_clip = tf.assign(x_noise, tf.clip_by_value(x_noise,
                                                    -noise_limit,
                                                    noise_limit))
```

噪声图像只是输入图像和对抗噪声的总和。

```
x_noisy_image = x_image + x_noise
```

把噪声图像添加到输入图像上时，它可能会溢出有效图像（像素）的边界，因此我们裁剪/限制噪声图像，确保它的像素值在0到1之间。

```
x_noisy_image = tf.clip_by_value(x_noisy_image, 0.0, 1.0)
```

卷积神经网络

我们会用PrettyTensor来构造卷积神经网络。首先需要将噪声图像的张量封装到PrettyTensor对象中，该对象提供了构造神经网络的函数。

```
x_pretty = pt.wrap(x_noisy_image)
```

将输入图像封装到PrettyTensor对象之后，用几行代码就能添加卷积层和全连接层。

```
with pt.defaults_scope(activation_fn=tf.nn.relu):
    y_pred, loss = x_pretty.\
        conv2d(kernel=5, depth=16, name='layer_conv1').\
        max_pool(kernel=2, stride=2).\
        conv2d(kernel=5, depth=36, name='layer_conv2').\
        max_pool(kernel=2, stride=2).\
        flatten().\
        fully_connected(size=128, name='layer_fc1').\
        softmax_classifier(num_classes=num_classes, labels=y_tru
e)
```

注意，在 `with` 代码块中， `pt.defaults_scope(activation_fn=tf.nn.relu)` 把 `activation_fn=tf.nn.relu` 当作每个的层参数，因此这些层都用到了Rectified Linear Units (ReLU) 。defaults_scope使我们能更方便地修改所有层的参数。

正常训练的优化器

这是会在常规优化程序里被训练的神经网络的变量列表。注意， `'x_noise:0'` 不在列表里，因此这个程序并不会优化对抗噪声 。

```
[var.name for var in tf.trainable_variables()]
```

```
['layer_conv1/weights:0',
 'layer_conv1/bias:0',
 'layer_conv2/weights:0',
 'layer_conv2/bias:0',
 'layer_fc1/weights:0',
 'layer_fc1/bias:0',
 'fully_connected/weights:0',
 'fully_connected/bias:0']
```

神经网络中这些变量的优化由Adam-optimizer完成，它用到上面PretyTensor构造的神经网络所返回的损失度量。

此时不执行优化，实际上这里根本没有计算，我们只是把优化对象添加到TensorFlow图表中，以便稍后运行。

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(
loss)
```

对抗噪声的优化器

获取变量列表，这些是需要在第二个程序里为对抗噪声做优化的变量。

```
adversary_variables = tf.get_collection(ADVERSARY_VARIABLES)
```

展示变量名称列表。这里只有一个元素，是我们在上面创建的对抗噪声变量。

```
[var.name for var in adversary_variables]
```

```
['x_noise:0']
```

我们会将常规优化的损失函数与所谓的L2-loss相结合。这将会得到在最佳分类准确率下的最小对抗噪声。

L2-loss由一个通常设置为接近零的权重缩放。

```
l2_loss_noise = noise_l2_weight * tf.nn.l2_loss(x_noise)
```

将正常的损失函数和对抗噪声的L2-loss相结合。

```
loss_adversary = loss + l2_loss_noise
```

现在可以为对抗噪声创建优化器。由于优化器并不能更新神经网络的所有变量，我们必须给出一个需要更新的变量的列表，即对抗噪声变量。注意，这里的学习率比上面的常规优化器要大很多。

```
optimizer_adversary = tf.train.AdamOptimizer(learning_rate=1e-2)
.minimize(loss_adversary, var_list=adversary_variables)
```

现在我们为神经网络创建了两个优化器，一个用于神经网络的变量，另一个用于对抗噪声的单个变量。

性能度量

在TensorFlow图表中，我们需要另外一些操作，以便在优化过程中向用户展示进度。

首先，计算出神经网络输出 `y_pred` 的预测类别号，它是一个包含10个元素的向量。类型号是最大元素的索引。

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

接着创建一个布尔数组，用来表示每张图像的预测类型是否与真实类型相同。

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

上面的计算先将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确度。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
))
```

运行TensorFlow

创建TensorFlow会话（session）

一旦创建了TensorFlow图，我们需要创建一个TensorFlow会话，用来运行图。

```
session = tf.Session()
```

初始化变量

我们需要在开始优化 `weights` 和 `biases` 变量之前对它们进行初始化。

```
session.run(tf.global_variables_initializer())
```

帮助函数将对抗噪声初始化/重置为零。

```
def init_noise():
    session.run(tf.variables_initializer([x_noise]))
```

调用函数来初始化对抗噪声。

```
init_noise()
```

用来优化迭代的帮助函数

在训练集中有55,000张图。用全部图像计算模型的梯度会花很多时间。因此我们在优化器的每次迭代里只用到了一小部分的图像。

如果内存耗尽导致电脑死机或变得很慢，你应该试着减少这些数量，但同时可能还需要更优化的迭代。

```
train_batch_size = 64
```

下面的函数用来执行一定数量的优化迭代，以此来逐渐改善神经网络的变量。在每次迭代中，会从训练集中选择新的一批数据，然后TensorFlow在这些训练样本上执行优化。每100次迭代会打印出进度。

这个函数与之前教程中的相似，除了现在它多了一个对抗目标类别(adversary target-class)的参数。当目标类别设为整数时，将会用它取代训练集中的真实类别号。也会用对抗优化器代替常规优化器，然后在每次优化之后，噪声将被限制/截断到允许的范围。这里优化了对抗噪声，并忽略神经网络中的其他变量。

```
def optimize(num_iterations, adversary_target_cls=None):
    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(num_iterations):

        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = data.train.next_batch(train_batc
h_size)

        # If we are searching for the adversarial noise, then
        # use the adversarial target-class instead.
        if adversary_target_cls is not None:
            # The class-labels are One-Hot encoded.

            # Set all the class-labels to zero.
            y_true_batch = np.zeros_like(y_true_batch)

            # Set the element for the adversarial target-class t
o 1.
            y_true_batch[:, adversary_target_cls] = 1.0

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # If doing normal optimization of the neural network.
        if adversary_target_cls is None:
            # Run the optimizer using this batch of training dat
a.

            # TensorFlow assigns the variables in feed_dict_train

            # to the placeholder variables and then runs the opt
imizer.
            session.run(optimizer, feed_dict=feed_dict_train)
        else:
```

```
            # Run the adversarial optimizer instead.
            # Note that we have 'faked' the class above to be
            # the adversarial target-class instead of the true c
lass.
            session.run(optimizer_adversary, feed_dict=feed_dict
_train)

            # Clip / limit the adversarial noise. This executes
            # another TensorFlow operation. It cannot be executed

            # in the same session.run() as the optimizer, because

            # it may run in parallel so the execution order is n
ot
            # guaranteed. We need the clip to run after the opti
mizer.
            session.run(x_noise_clip)

        # Print status every 100 iterations.
        if (i % 100 == 0) or (i == num_iterations - 1):
            # Calculate the accuracy on the training-set.
            acc = session.run(accuracy, feed_dict=feed_dict_trai
n)

            # Message for printing.
            msg = "Optimization Iteration: {0:>6}, Training Accu
racy: {1:>6.1%}"

            # Print it.
            print(msg.format(i, acc))

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_
dif)))))
```

获取及绘制噪声的帮助函数

这个函数从TensorFlow图表中获取对抗噪声。

```
def get_noise():
    # Run the TensorFlow session to retrieve the contents of
    # the x_noise variable inside the graph.
    noise = session.run(x_noise)

    return np.squeeze(noise)
```

这个函数绘制了对抗噪声，并打印一些统计信息。

```
def plot_noise():
    # Get the adversarial noise from inside the TensorFlow graph.

    noise = get_noise()

    # Print statistics.
    print("Noise:")
    print("- Min:", noise.min())
    print("- Max:", noise.max())
    print("- Std:", noise.std())

    # Plot the noise.
    plt.imshow(noise, interpolation='nearest', cmap='seismic',
               vmin=-1.0, vmax=1.0)
```

用来绘制错误样本的帮助函数

函数用来绘制测试集中被误分类的样本。

```
def plot_example_errors(cls_pred, correct):
    # This function is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # correct is a boolean array whether the predicted class
    # is equal to the true class for each image in the test-set.

    # Negate the boolean array.
    incorrect = (correct == False)

    # Get the images from the test-set that have been
    # incorrectly classified.
    images = data.test.images[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = data.test.cls[incorrect]

    # Get the adversarial noise from inside the TensorFlow graph.

    noise = get_noise()

    # Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9],
                noise=noise)
```

绘制混淆（confusion）矩阵的帮助函数

```
def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the true classifications for the test-set.
    cls_true = data.test.cls

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_true,
                          y_pred=cls_pred)

    # Print the confusion matrix as text.
    print(cm)
```

展示性能的帮助函数

函数用来打印测试集上的分类准确度。

为测试集上的所有图片计算分类会花费一段时间，因此我们直接用这个函数来调用上面的结果，这样就不用每次都重新计算了。

这个函数可能会占用很多电脑内存，这也是为什么将测试集分成更小的几个部分。如果你的电脑内存比较小或死机了，就要试着降低batch-size。

```
# Split the test-set into smaller batches of this size.
test_batch_size = 256

def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # Number of images in the test-set.
    num_test = len(data.test.images)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_test, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing thi
s.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_test:
        # The ending index for the next batch is denoted j.
        j = min(i + test_batch_size, num_test)
```

```
        # Get the images from the test-set between index i and j.

        images = data.test.images[i:j, :]

        # Get the associated labels.
        labels = data.test.labels[i:j, :]

        # Create a feed-dict with these images and labels.
        feed_dict = {x: images,
                     y_true: labels}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_d
ict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Convenience variable for the true class-numbers of the tes
t-set.
    cls_true = data.test.cls

    # Create a boolean array whether each image is correctly cla
ssified.
    correct = (cls_true == cls_pred)

    # Calculate the number of correctly classified images.
    # When summing a boolean array, False means 0 and True means
 1.
    correct_sum = correct.sum()

    # Classification accuracy is the number of correctly classif
ied
    # images divided by the total number of images in the test-s
et.
    acc = float(correct_sum) / num_test

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, correct_sum, num_test))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```



神经网络的常规优化

此时对抗噪声还没有效果，因为上面只将它初始化为零，在优化过程中并未更新。

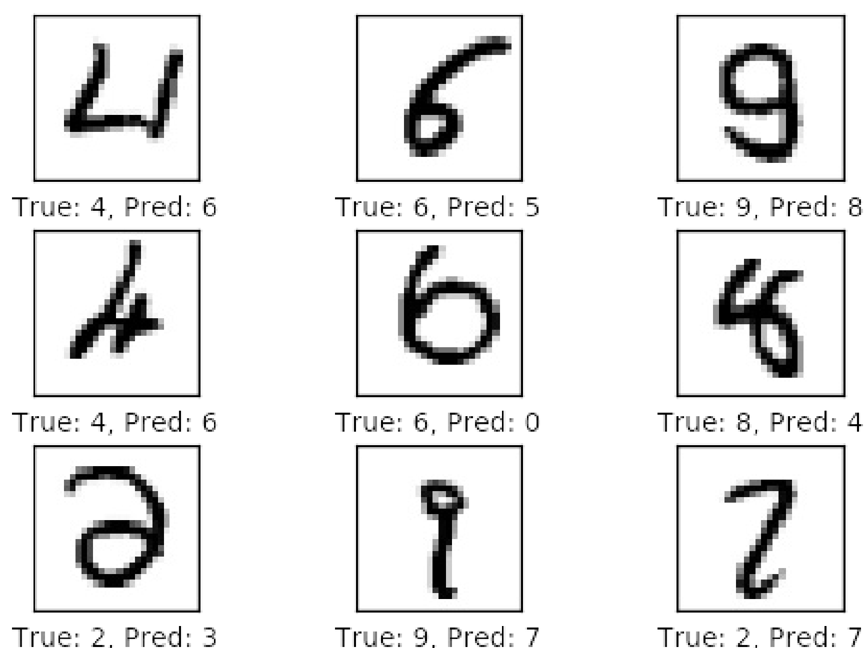
```
optimize(num_iterations=1000)
```

```
Optimization Iteration:      0, Training Accuracy:  12.5%
Optimization Iteration:    100, Training Accuracy:  90.6%
Optimization Iteration:    200, Training Accuracy:  84.4%
Optimization Iteration:    300, Training Accuracy:  84.4%
Optimization Iteration:    400, Training Accuracy:  89.1%
Optimization Iteration:    500, Training Accuracy:  87.5%
Optimization Iteration:    600, Training Accuracy:  93.8%
Optimization Iteration:    700, Training Accuracy:  93.8%
Optimization Iteration:    800, Training Accuracy:  93.8%
Optimization Iteration:    900, Training Accuracy:  96.9%
Optimization Iteration:    999, Training Accuracy:  92.2%
Time usage: 0:00:03
```

测试集上的分类准确率大约96-97%。（每次运行Python Notobook时，结果会有所变化。）

```
print_test_accuracy(show_example_errors=True)
```

```
Accuracy on Test-Set: 96.3% (9633 / 10000)
Example errors:
```



寻找对抗噪声

在我们开始优化对抗噪声之前，先将它初始化为零。上面已经完成了这一步，但这里再执行一次，以防你用其他目标类型重新运行代码。

```
init_noise()
```

现在执行对抗噪声的优化。这里使用对抗优化器而不是常规优化器，这说明它只优化对抗噪声变量，同时忽略神经网络中的其他变量。

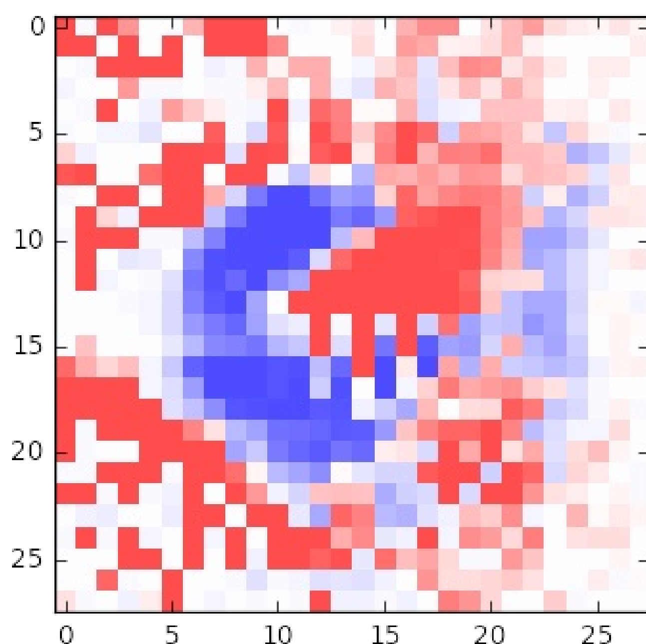
```
optimize(num_iterations=1000, adversary_target_cls=3)
```

```
Optimization Iteration:      0, Training Accuracy:   6.2%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    200, Training Accuracy:  96.9%
Optimization Iteration:    300, Training Accuracy:  98.4%
Optimization Iteration:    400, Training Accuracy:  95.3%
Optimization Iteration:    500, Training Accuracy:  96.9%
Optimization Iteration:    600, Training Accuracy: 100.0%
Optimization Iteration:    700, Training Accuracy:  98.4%
Optimization Iteration:    800, Training Accuracy:  95.3%
Optimization Iteration:    900, Training Accuracy:  93.8%
Optimization Iteration:    999, Training Accuracy: 100.0%
Time usage: 0:00:03
```

现在对抗噪声已经被优化了，可以在一张图像中展示出来。红色像素显示了正噪声值，蓝色像素显示了负噪声值。这个噪声模式将会被添加到每张输入图像中。正噪声值（红）使像素变暗，负噪声值（蓝）使像素变亮。如下所示。

```
plot_noise()
```

```
Noise:
- Min: -0.35
- Max: 0.35
- Std: 0.195455
```



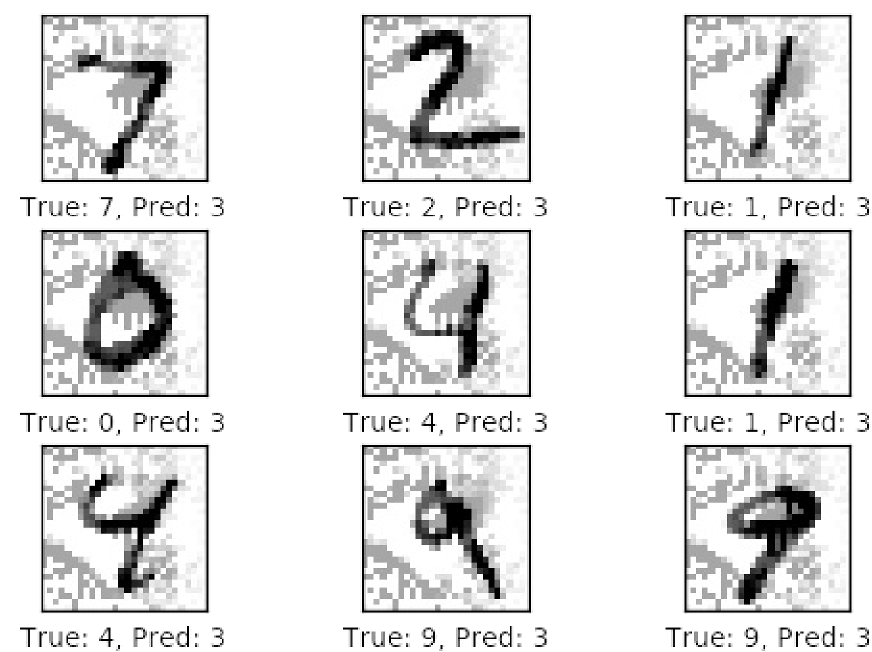
当测试集的所有图像上都添加了该噪声之后，根据选定的目标类别，分类准确率通常在是10-15%之间。我们也能从混淆矩阵中看出，测试集中的大多数图像都被分类成期望的目标类别——尽管有些目标类型比其他的需要更多的对抗噪声。

所以我们找到了使对抗噪声，使神经网络将测试集中绝大部分图像误分类成期望的类别。

我们也可以画出一些带有对抗噪声的误分类图像样本。噪声清晰可见，但人眼还是可以轻易地分辨出数字。

```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

```
Accuracy on Test-Set: 13.2% (1323 / 10000)
Example errors:
```




```
Confusion Matrix:
[[  85    0    0  895    0    0    0    0    0    0]
 [   0    0    0 1135    0    0    0    0    0    0]
 [   0    0   46  986    0    0    0    0    0    0]
 [   0    0    0 1010    0    0    0    0    0    0]
 [   0    0    0  959   20    0    0    0    3    0]
 [   0    0    0  847    0   45    0    0    0    0]
 [   0    0    0  914    0    1   42    0    1    0]
 [   0    0    0  977    0    0    0   51    0    0]
 [   0    0    0  952    0    0    0    0   22    0]
 [   0    0    1 1006    0    0    0    0    0    2]]
```

所有目标类别的对抗噪声

这是帮助函数用于寻找所有目标类别的对抗噪声。函数从类型号0遍历到9，执行上面的优化。然后将结果保存到一个数组中。

```
def find_all_noise(num_iterations=1000):
    # Adversarial noise for all target-classes.
    all_noise = []

    # For each target-class.
    for i in range(num_classes):
        print("Finding adversarial noise for target-class:", i)

        # Reset the adversarial noise to zero.
        init_noise()

        # Optimize the adversarial noise.
        optimize(num_iterations=num_iterations,
                 adversary_target_cls=i)

        # Get the adversarial noise from inside the TensorFlow g
raph.
        noise = get_noise()

        # Append the noise to the array.
        all_noise.append(noise)

        # Print newline.
        print()

    return all_noise
```

```
all_noise = find_all_noise(num_iterations=300)
```

```
Finding adversarial noise for target-class: 0
Optimization Iteration:      0, Training Accuracy:   9.4%
Optimization Iteration:    100, Training Accuracy:  90.6%
Optimization Iteration:    200, Training Accuracy:  92.2%
Optimization Iteration:    299, Training Accuracy:  93.8%
Time usage: 0:00:01

Finding adversarial noise for target-class: 1
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  62.5%
Optimization Iteration:    200, Training Accuracy:  62.5%
Optimization Iteration:    299, Training Accuracy:  75.0%
Time usage: 0:00:01

Finding adversarial noise for target-class: 2
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    200, Training Accuracy:  95.3%
Optimization Iteration:    299, Training Accuracy:  96.9%
Time usage: 0:00:01
```

```
Finding adversarial noise for target-class: 3
Optimization Iteration:      0, Training Accuracy:   6.2%
Optimization Iteration:    100, Training Accuracy:  98.4%
Optimization Iteration:    200, Training Accuracy:  96.9%
Optimization Iteration:    299, Training Accuracy:  98.4%
Time usage: 0:00:01

Finding adversarial noise for target-class: 4
Optimization Iteration:      0, Training Accuracy:  12.5%
Optimization Iteration:    100, Training Accuracy:  81.2%
Optimization Iteration:    200, Training Accuracy:  82.8%
Optimization Iteration:    299, Training Accuracy:  82.8%
Time usage: 0:00:01

Finding adversarial noise for target-class: 5
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  96.9%
Optimization Iteration:    200, Training Accuracy:  96.9%
Optimization Iteration:    299, Training Accuracy:  98.4%
Time usage: 0:00:01

Finding adversarial noise for target-class: 6
Optimization Iteration:      0, Training Accuracy:   6.2%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    200, Training Accuracy:  92.2%
Optimization Iteration:    299, Training Accuracy:  96.9%
Time usage: 0:00:01

Finding adversarial noise for target-class: 7
Optimization Iteration:      0, Training Accuracy:  12.5%
Optimization Iteration:    100, Training Accuracy:  98.4%
Optimization Iteration:    200, Training Accuracy:  93.8%
Optimization Iteration:    299, Training Accuracy:  92.2%
Time usage: 0:00:01

Finding adversarial noise for target-class: 8
Optimization Iteration:      0, Training Accuracy:   4.7%
Optimization Iteration:    100, Training Accuracy:  96.9%
Optimization Iteration:    200, Training Accuracy:  93.8%
Optimization Iteration:    299, Training Accuracy:  96.9%
Time usage: 0:00:01

Finding adversarial noise for target-class: 9
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  84.4%
Optimization Iteration:    200, Training Accuracy:  87.5%
Optimization Iteration:    299, Training Accuracy:  90.6%
Time usage: 0:00:01
```

绘制所有目标类型的对抗噪声

这个帮助函数用于在栅格中绘制所有目标类型（0到9）的对抗噪声。

```
def plot_all_noise(all_noise):
    # Create figure with 10 sub-plots.
    fig, axes = plt.subplots(2, 5)
    fig.subplots_adjust(hspace=0.2, wspace=0.1)

    # For each sub-plot.
    for i, ax in enumerate(axes.flat):
        # Get the adversarial noise for the i'th target-class.
        noise = all_noise[i]

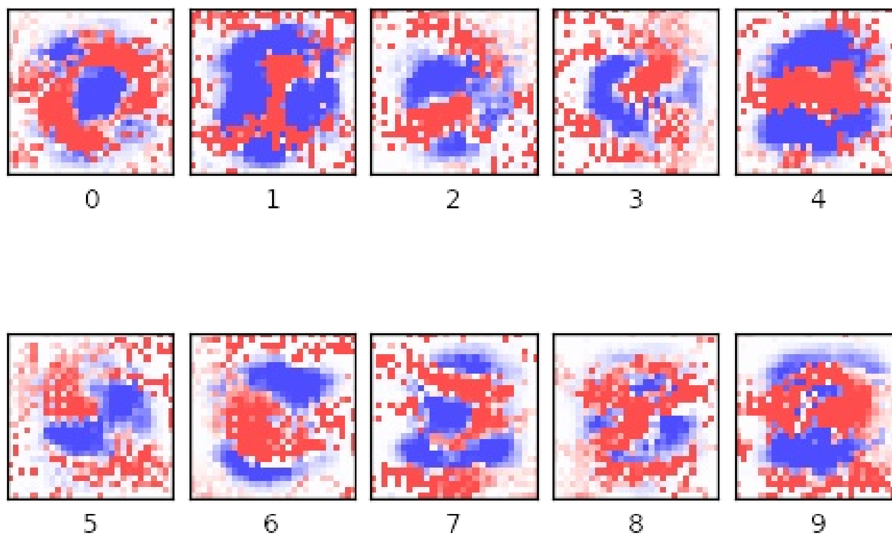
        # Plot the noise.
        ax.imshow(noise,
                  cmap='seismic', interpolation='nearest',
                  vmin=-1.0, vmax=1.0)

        # Show the classes as the label on the x-axis.
        ax.set_xlabel(i)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

```
plot_all_noise(all_noise)
```



红色像素显示正噪声值，蓝色像素显示负噪声值。

在其中一些噪声图像中，你可以看到数字的痕迹。例如，目标类型0的噪声显示了一个被蓝色包围的红圈。这说明会以圆形状将一些噪声添加到图像中，并抑制其他像素。这足以让MNIST数据集中的大部分图像被误分类成0。另外一个例子是3的噪声，图像的红色像素也显示了数字3的痕迹。但其他类别的噪声不太明显。

对抗噪声的免疫

现在试着让神经网络对对抗噪声免疫。我们重新训练神经网络，使其忽略对抗噪声。这个过程可以重复多次。

帮助函数创建了对对抗噪声免疫的神经网络

这是使神经网络对对抗噪声免疫的帮助函数。首先运行优化来找到对抗噪声。接着执行常规优化使神经网络对该噪声免疫。

```
def make_immune(target_cls, num_iterations_adversary=500,
                num_iterations_immune=200):

    print("Target-class:", target_cls)
    print("Finding adversarial noise ...")

    # Find the adversarial noise.
    optimize(num_iterations=num_iterations_adversary,
             adversary_target_cls=target_cls)

    # Newline.
    print()

    # Print classification accuracy.
    print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False)

    # Newline.
    print()

    print("Making the neural network immune to the noise ...")

    # Try and make the neural network immune to this noise.
    # Note that the adversarial noise has not been reset to zero
    # so the x_noise variable still holds the noise.
    # So we are training the neural network to ignore the noise.
    optimize(num_iterations=num_iterations_immune)

    # Newline.
    print()

    # Print classification accuracy.
    print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False)
```

对目标类型3的噪声免疫

首先尝试使神经网络对目标类型3的对抗噪声免疫。

我们先找到导致神经网络误分类测试集上大多数图像的对抗噪声。接着执行常规优化，其变量经过微调从而忽略噪声，使得分类准确率再次达到95-97%。

```
make_immune(target_cls=3)
```

```
Target-class: 3
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   3.1%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    200, Training Accuracy:  93.8%
Optimization Iteration:    300, Training Accuracy:  96.9%
Optimization Iteration:    400, Training Accuracy:  96.9%
Optimization Iteration:    499, Training Accuracy:  96.9%
Time usage: 0:00:02

Accuracy on Test-Set: 14.4% (1443 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  42.2%
Optimization Iteration:    100, Training Accuracy:  90.6%
Optimization Iteration:    199, Training Accuracy:  89.1%
Time usage: 0:00:01

Accuracy on Test-Set: 95.3% (9529 / 10000)
```

现在试着再次运行它。 现在更难为目标类别3找到对抗噪声 。神经网络似乎已经变得对对抗噪声有些免疫。

```
make_immune(target_cls=3)
```

```
Target-class: 3
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  32.8%
Optimization Iteration:    200, Training Accuracy:  32.8%
Optimization Iteration:    300, Training Accuracy:  29.7%
Optimization Iteration:    400, Training Accuracy:  34.4%
Optimization Iteration:    499, Training Accuracy:  26.6%
Time usage: 0:00:02

Accuracy on Test-Set: 72.1% (7207 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  75.0%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    199, Training Accuracy:  92.2%
Time usage: 0:00:01

Accuracy on Test-Set: 95.2% (9519 / 10000)
```

对所有目标类型的噪声免疫

现在，试着使神经网络对所有目标类型的噪声免疫。不幸的是，看起来并不太好。

```
for i in range(10):
    make_immune(target_cls=i)

    # Print newline.
    print()
```

```
Target-class: 0
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   4.7%
Optimization Iteration:    100, Training Accuracy:  73.4%
Optimization Iteration:    200, Training Accuracy:  75.0%
Optimization Iteration:    300, Training Accuracy:  85.9%
Optimization Iteration:    400, Training Accuracy:  81.2%
Optimization Iteration:    499, Training Accuracy:  90.6%
Time usage: 0:00:02

Accuracy on Test-Set: 23.3% (2326 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  34.4%
Optimization Iteration:    100, Training Accuracy:  95.3%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 95.6% (9559 / 10000)

Target-class: 1
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:  12.5%
Optimization Iteration:    100, Training Accuracy:  57.8%
Optimization Iteration:    200, Training Accuracy:  62.5%
Optimization Iteration:    300, Training Accuracy:  62.5%
Optimization Iteration:    400, Training Accuracy:  67.2%
Optimization Iteration:    499, Training Accuracy:  67.2%
Time usage: 0:00:02

Accuracy on Test-Set: 42.2% (4218 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  59.4%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 95.5% (9555 / 10000)

Target-class: 2
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   6.2%
```

```
Optimization Iteration:    100, Training Accuracy:  43.8%
Optimization Iteration:    200, Training Accuracy:  57.8%
Optimization Iteration:    300, Training Accuracy:  70.3%
Optimization Iteration:    400, Training Accuracy:  68.8%
Optimization Iteration:    499, Training Accuracy:  71.9%
Time usage: 0:00:02

Accuracy on Test-Set: 46.4% (4639 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  59.4%
Optimization Iteration:    100, Training Accuracy:  96.9%
Optimization Iteration:    199, Training Accuracy:  92.2%
Time usage: 0:00:01

Accuracy on Test-Set: 95.5% (9545 / 10000)

Target-class: 3
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   6.2%
Optimization Iteration:    100, Training Accuracy:  48.4%
Optimization Iteration:    200, Training Accuracy:  46.9%
Optimization Iteration:    300, Training Accuracy:  53.1%
Optimization Iteration:    400, Training Accuracy:  50.0%
Optimization Iteration:    499, Training Accuracy:  48.4%
Time usage: 0:00:02

Accuracy on Test-Set: 56.5% (5648 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  54.7%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    199, Training Accuracy:  96.9%
Time usage: 0:00:01

Accuracy on Test-Set: 95.8% (9581 / 10000)

Target-class: 4
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   9.4%
Optimization Iteration:    100, Training Accuracy:  85.9%
Optimization Iteration:    200, Training Accuracy:  85.9%
Optimization Iteration:    300, Training Accuracy:  87.5%
Optimization Iteration:    400, Training Accuracy:  95.3%
Optimization Iteration:    499, Training Accuracy:  92.2%
Time usage: 0:00:02

Accuracy on Test-Set: 15.6% (1557 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  18.8%
Optimization Iteration:    100, Training Accuracy:  95.3%
Optimization Iteration:    199, Training Accuracy:  96.9%
```

```
Time usage: 0:00:01

Accuracy on Test-Set: 95.6% (9557 / 10000)

Target-class: 5
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:  18.8%
Optimization Iteration:    100, Training Accuracy:  71.9%
Optimization Iteration:    200, Training Accuracy:  90.6%
Optimization Iteration:    300, Training Accuracy:  95.3%
Optimization Iteration:    400, Training Accuracy:  89.1%
Optimization Iteration:    499, Training Accuracy:  92.2%
Time usage: 0:00:02

Accuracy on Test-Set: 17.4% (1745 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  15.6%
Optimization Iteration:    100, Training Accuracy:  96.9%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 96.0% (9601 / 10000)

Target-class: 6
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:  10.9%
Optimization Iteration:    100, Training Accuracy:  81.2%
Optimization Iteration:    200, Training Accuracy:  93.8%
Optimization Iteration:    300, Training Accuracy:  92.2%
Optimization Iteration:    400, Training Accuracy:  89.1%
Optimization Iteration:    499, Training Accuracy:  92.2%
Time usage: 0:00:02

Accuracy on Test-Set: 17.6% (1762 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  20.3%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 95.7% (9570 / 10000)

Target-class: 7
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:  14.1%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    200, Training Accuracy:  98.4%
Optimization Iteration:    300, Training Accuracy: 100.0%
Optimization Iteration:    400, Training Accuracy:  96.9%
Optimization Iteration:    499, Training Accuracy: 100.0%
Time usage: 0:00:02
```

```
Accuracy on Test-Set: 12.8% (1281 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  12.5%
Optimization Iteration:    100, Training Accuracy:  98.4%
Optimization Iteration:    199, Training Accuracy:  98.4%
Time usage: 0:00:01

Accuracy on Test-Set: 95.9% (9587 / 10000)

Target-class: 8
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   4.7%
Optimization Iteration:    100, Training Accuracy:  64.1%
Optimization Iteration:    200, Training Accuracy:  81.2%
Optimization Iteration:    300, Training Accuracy:  71.9%
Optimization Iteration:    400, Training Accuracy:  78.1%
Optimization Iteration:    499, Training Accuracy:  84.4%
Time usage: 0:00:02

Accuracy on Test-Set: 24.9% (2493 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  25.0%
Optimization Iteration:    100, Training Accuracy:  95.3%
Optimization Iteration:    199, Training Accuracy:  96.9%
Time usage: 0:00:01

Accuracy on Test-Set: 96.0% (9601 / 10000)

Target-class: 9
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   9.4%
Optimization Iteration:    100, Training Accuracy:  48.4%
Optimization Iteration:    200, Training Accuracy:  50.0%
Optimization Iteration:    300, Training Accuracy:  53.1%
Optimization Iteration:    400, Training Accuracy:  64.1%
Optimization Iteration:    499, Training Accuracy:  65.6%
Time usage: 0:00:02

Accuracy on Test-Set: 45.5% (4546 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  51.6%
Optimization Iteration:    100, Training Accuracy:  95.3%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 96.2% (9615 / 10000)
```

对所有目标类别免疫（执行两次）

现在试着执行两次，使神经网络对所有目标类别的噪声免疫。不幸的是，结果也不太好。

使神经网络免受一个对抗目标类型的影响，似乎使得它对另外一个目标类型失去了免疫。

```
for i in range(10):
    make_immune(target_cls=i)

    # Print newline.
    print()

    make_immune(target_cls=i)

    # Print newline.
    print()
```

```
Target-class: 0
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  53.1%
Optimization Iteration:    200, Training Accuracy:  73.4%
Optimization Iteration:    300, Training Accuracy:  79.7%
Optimization Iteration:    400, Training Accuracy:  84.4%
Optimization Iteration:    499, Training Accuracy:  95.3%
Time usage: 0:00:02

Accuracy on Test-Set: 29.2% (2921 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  29.7%
Optimization Iteration:    100, Training Accuracy:  96.9%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 96.2% (9619 / 10000)

Target-class: 0
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   1.6%
Optimization Iteration:    100, Training Accuracy:  12.5%
Optimization Iteration:    200, Training Accuracy:   7.8%
Optimization Iteration:    300, Training Accuracy:  18.8%
Optimization Iteration:    400, Training Accuracy:   9.4%
Optimization Iteration:    499, Training Accuracy:   9.4%
Time usage: 0:00:02

Accuracy on Test-Set: 94.4% (9437 / 10000)
```

```
Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  89.1%
Optimization Iteration:    100, Training Accuracy:  98.4%
Optimization Iteration:    199, Training Accuracy:  93.8%
Time usage: 0:00:01

Accuracy on Test-Set: 96.4% (9635 / 10000)

Target-class: 1
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  42.2%
Optimization Iteration:    200, Training Accuracy:  60.9%
Optimization Iteration:    300, Training Accuracy:  75.0%
Optimization Iteration:    400, Training Accuracy:  70.3%
Optimization Iteration:    499, Training Accuracy:  85.9%
Time usage: 0:00:02

Accuracy on Test-Set: 28.7% (2875 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  39.1%
Optimization Iteration:    100, Training Accuracy:  98.4%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 96.4% (9643 / 10000)

Target-class: 1
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  15.6%
Optimization Iteration:    200, Training Accuracy:  18.8%
Optimization Iteration:    300, Training Accuracy:  12.5%
Optimization Iteration:    400, Training Accuracy:   9.4%
Optimization Iteration:    499, Training Accuracy:  12.5%
Time usage: 0:00:02

Accuracy on Test-Set: 94.3% (9428 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  95.3%
Optimization Iteration:    100, Training Accuracy:  95.3%
Optimization Iteration:    199, Training Accuracy:  92.2%
Time usage: 0:00:01

Accuracy on Test-Set: 96.9% (9685 / 10000)

Target-class: 2
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   6.2%
Optimization Iteration:    100, Training Accuracy:  60.9%
```

```
Optimization Iteration:    200, Training Accuracy:  64.1%
Optimization Iteration:    300, Training Accuracy:  71.9%
Optimization Iteration:    400, Training Accuracy:  75.0%
Optimization Iteration:    499, Training Accuracy:  82.8%
Time usage: 0:00:02

Accuracy on Test-Set: 34.3% (3427 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  31.2%
Optimization Iteration:    100, Training Accuracy: 100.0%
Optimization Iteration:    199, Training Accuracy:  98.4%
Time usage: 0:00:01

Accuracy on Test-Set: 96.6% (9657 / 10000)

Target-class: 2
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   6.2%
Optimization Iteration:    100, Training Accuracy:   9.4%
Optimization Iteration:    200, Training Accuracy:  14.1%
Optimization Iteration:    300, Training Accuracy:  10.9%
Optimization Iteration:    400, Training Accuracy:   7.8%
Optimization Iteration:    499, Training Accuracy:  17.2%
Time usage: 0:00:02

Accuracy on Test-Set: 94.3% (9435 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  96.9%
Optimization Iteration:    100, Training Accuracy:  98.4%
Optimization Iteration:    199, Training Accuracy:  96.9%
Time usage: 0:00:01

Accuracy on Test-Set: 96.6% (9664 / 10000)

Target-class: 3
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:  14.1%
Optimization Iteration:    100, Training Accuracy:  20.3%
Optimization Iteration:    200, Training Accuracy:  40.6%
Optimization Iteration:    300, Training Accuracy:  57.8%
Optimization Iteration:    400, Training Accuracy:  54.7%
Optimization Iteration:    499, Training Accuracy:  64.1%
Time usage: 0:00:02

Accuracy on Test-Set: 48.4% (4837 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  54.7%
Optimization Iteration:    100, Training Accuracy:  98.4%
Optimization Iteration:    199, Training Accuracy: 100.0%
Time usage: 0:00:01
```

```
Accuracy on Test-Set: 96.5% (9650 / 10000)

Target-class: 3
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   4.7%
Optimization Iteration:    100, Training Accuracy:  10.9%
Optimization Iteration:    200, Training Accuracy:  17.2%
Optimization Iteration:    300, Training Accuracy:  15.6%
Optimization Iteration:    400, Training Accuracy:   1.6%
Optimization Iteration:    499, Training Accuracy:   9.4%
Time usage: 0:00:02

Accuracy on Test-Set: 95.7% (9570 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  95.3%
Optimization Iteration:    100, Training Accuracy:  90.6%
Optimization Iteration:    199, Training Accuracy:  98.4%
Time usage: 0:00:01

Accuracy on Test-Set: 96.7% (9667 / 10000)

Target-class: 4
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  67.2%
Optimization Iteration:    200, Training Accuracy:  78.1%
Optimization Iteration:    300, Training Accuracy:  79.7%
Optimization Iteration:    400, Training Accuracy:  81.2%
Optimization Iteration:    499, Training Accuracy:  96.9%
Time usage: 0:00:02

Accuracy on Test-Set: 23.7% (2373 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  26.6%
Optimization Iteration:    100, Training Accuracy:  95.3%
Optimization Iteration:    199, Training Accuracy:  96.9%
Time usage: 0:00:01

Accuracy on Test-Set: 96.3% (9632 / 10000)

Target-class: 4
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   4.7%
Optimization Iteration:    100, Training Accuracy:   7.8%
Optimization Iteration:    200, Training Accuracy:  12.5%
Optimization Iteration:    300, Training Accuracy:  15.6%
Optimization Iteration:    400, Training Accuracy:   7.8%
Optimization Iteration:    499, Training Accuracy:  14.1%
Time usage: 0:00:02
```

```
Accuracy on Test-Set: 92.0% (9197 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  92.2%
Optimization Iteration:    100, Training Accuracy:  95.3%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 96.3% (9632 / 10000)

Target-class: 5
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   4.7%
Optimization Iteration:    100, Training Accuracy:  57.8%
Optimization Iteration:    200, Training Accuracy:  76.6%
Optimization Iteration:    300, Training Accuracy:  85.9%
Optimization Iteration:    400, Training Accuracy:  89.1%
Optimization Iteration:    499, Training Accuracy:  85.9%
Time usage: 0:00:02

Accuracy on Test-Set: 23.0% (2297 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  28.1%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    199, Training Accuracy:  98.4%
Time usage: 0:00:01

Accuracy on Test-Set: 96.6% (9663 / 10000)

Target-class: 5
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   6.2%
Optimization Iteration:    100, Training Accuracy:  10.9%
Optimization Iteration:    200, Training Accuracy:  18.8%
Optimization Iteration:    300, Training Accuracy:  18.8%
Optimization Iteration:    400, Training Accuracy:  20.3%
Optimization Iteration:    499, Training Accuracy:  21.9%
Time usage: 0:00:02

Accuracy on Test-Set: 88.2% (8824 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  93.8%
Optimization Iteration:    100, Training Accuracy:  93.8%
Optimization Iteration:    199, Training Accuracy:  93.8%
Time usage: 0:00:01

Accuracy on Test-Set: 96.7% (9665 / 10000)

Target-class: 6
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   7.8%
```

```
Optimization Iteration:    100, Training Accuracy:  40.6%
Optimization Iteration:    200, Training Accuracy:  53.1%
Optimization Iteration:    300, Training Accuracy:  51.6%
Optimization Iteration:    400, Training Accuracy:  56.2%
Optimization Iteration:    499, Training Accuracy:  62.5%
Time usage: 0:00:02

Accuracy on Test-Set: 44.0% (4400 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  39.1%
Optimization Iteration:    100, Training Accuracy:  96.9%
Optimization Iteration:    199, Training Accuracy:  93.8%
Time usage: 0:00:01

Accuracy on Test-Set: 96.4% (9642 / 10000)

Target-class: 6
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   4.7%
Optimization Iteration:    100, Training Accuracy:  17.2%
Optimization Iteration:    200, Training Accuracy:  12.5%
Optimization Iteration:    300, Training Accuracy:  14.1%
Optimization Iteration:    400, Training Accuracy:  20.3%
Optimization Iteration:    499, Training Accuracy:   7.8%
Time usage: 0:00:02

Accuracy on Test-Set: 94.6% (9457 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  93.8%
Optimization Iteration:    100, Training Accuracy: 100.0%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 96.8% (9682 / 10000)

Target-class: 7
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   4.7%
Optimization Iteration:    100, Training Accuracy:  65.6%
Optimization Iteration:    200, Training Accuracy:  89.1%
Optimization Iteration:    300, Training Accuracy:  82.8%
Optimization Iteration:    400, Training Accuracy:  85.9%
Optimization Iteration:    499, Training Accuracy:  90.6%
Time usage: 0:00:02

Accuracy on Test-Set: 18.1% (1809 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  23.4%
Optimization Iteration:    100, Training Accuracy:  95.3%
Optimization Iteration:    199, Training Accuracy:  93.8%
```

```
Time usage: 0:00:01

Accuracy on Test-Set: 96.8% (9682 / 10000)

Target-class: 7
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:  12.5%
Optimization Iteration:    100, Training Accuracy:  10.9%
Optimization Iteration:    200, Training Accuracy:  18.8%
Optimization Iteration:    300, Training Accuracy:  18.8%
Optimization Iteration:    400, Training Accuracy:  28.1%
Optimization Iteration:    499, Training Accuracy:  18.8%
Time usage: 0:00:02

Accuracy on Test-Set: 84.1% (8412 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  84.4%
Optimization Iteration:    100, Training Accuracy: 100.0%
Optimization Iteration:    199, Training Accuracy: 100.0%
Time usage: 0:00:01

Accuracy on Test-Set: 97.0% (9699 / 10000)

Target-class: 8
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  48.4%
Optimization Iteration:    200, Training Accuracy:  46.9%
Optimization Iteration:    300, Training Accuracy:  71.9%
Optimization Iteration:    400, Training Accuracy:  70.3%
Optimization Iteration:    499, Training Accuracy:  75.0%
Time usage: 0:00:02

Accuracy on Test-Set: 36.8% (3678 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  48.4%
Optimization Iteration:    100, Training Accuracy:  96.9%
Optimization Iteration:    199, Training Accuracy:  93.8%
Time usage: 0:00:01

Accuracy on Test-Set: 97.0% (9699 / 10000)

Target-class: 8
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   7.8%
Optimization Iteration:    100, Training Accuracy:  14.1%
Optimization Iteration:    200, Training Accuracy:  12.5%
Optimization Iteration:    300, Training Accuracy:   7.8%
Optimization Iteration:    400, Training Accuracy:   4.7%
Optimization Iteration:    499, Training Accuracy:   9.4%
Time usage: 0:00:02
```

```
Accuracy on Test-Set: 96.2% (9625 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  96.9%
Optimization Iteration:    100, Training Accuracy:  98.4%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

Accuracy on Test-Set: 97.2% (9720 / 10000)

Target-class: 9
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   9.4%
Optimization Iteration:    100, Training Accuracy:  23.4%
Optimization Iteration:    200, Training Accuracy:  43.8%
Optimization Iteration:    300, Training Accuracy:  37.5%
Optimization Iteration:    400, Training Accuracy:  45.3%
Optimization Iteration:    499, Training Accuracy:  39.1%
Time usage: 0:00:02

Accuracy on Test-Set: 64.9% (6494 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  67.2%
Optimization Iteration:    100, Training Accuracy:  95.3%
Optimization Iteration:    199, Training Accuracy:  98.4%
Time usage: 0:00:01

Accuracy on Test-Set: 97.5% (9746 / 10000)

Target-class: 9
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   9.4%
Optimization Iteration:    100, Training Accuracy:   7.8%
Optimization Iteration:    200, Training Accuracy:  10.9%
Optimization Iteration:    300, Training Accuracy:  15.6%
Optimization Iteration:    400, Training Accuracy:  12.5%
Optimization Iteration:    499, Training Accuracy:   4.7%
Time usage: 0:00:02

Accuracy on Test-Set: 97.1% (9709 / 10000)

Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  98.4%
Optimization Iteration:    100, Training Accuracy: 100.0%
Optimization Iteration:    199, Training Accuracy:  95.3%
Time usage: 0:00:01

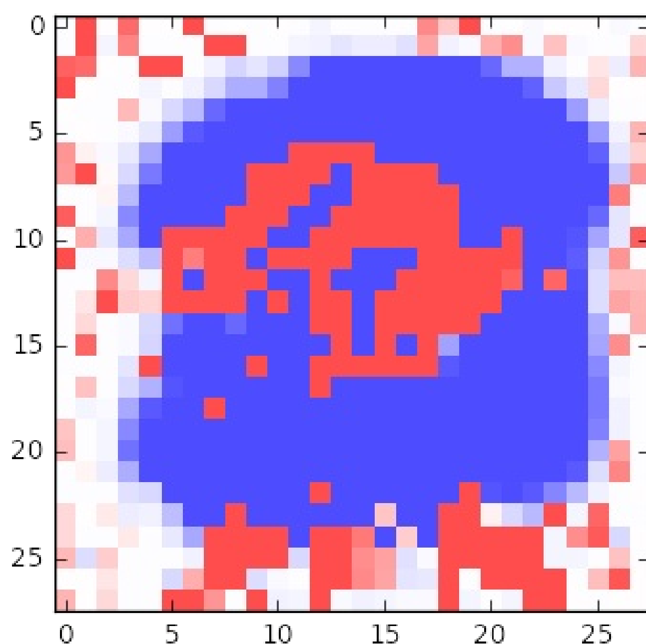
Accuracy on Test-Set: 97.7% (9768 / 10000)
```

绘制对抗噪声

现在我们已经对神经网络和对抗网络都进行了很多优化。让我们看看对抗噪声长什么样。

```
plot_noise()
```

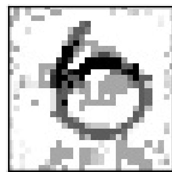
```
Noise:
- Min: -0.35
- Max: 0.35
- Std: 0.270488
```

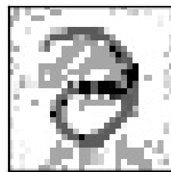


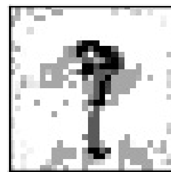
有趣的是，相比优化之前的干净图像，神经网络在噪声图像上有更高的分类准确率。

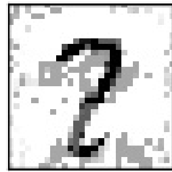
```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

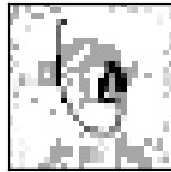
```
Accuracy on Test-Set: 97.7% (9768 / 10000)
Example errors:
```

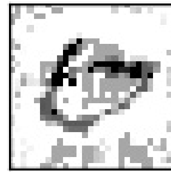

True: 6, Pred: 0

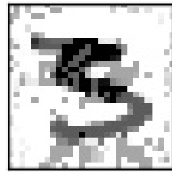

True: 2, Pred: 3

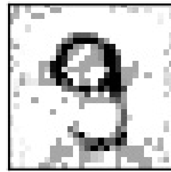

True: 9, Pred: 1

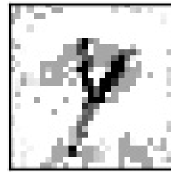

True: 2, Pred: 7


True: 6, Pred: 4


True: 6, Pred: 5


True: 3, Pred: 5


True: 9, Pred: 8


True: 4, Pred: 1

```
Confusion Matrix:
[[ 972    0    1    0    0    0    2    1    3    1]
 [   0 1119    4    0    0    2    2    0    8    0]
 [   3    0 1006    9    1    1    1    5    4    2]
 [   1    0    1  997    0    5    0    4    2    0]
 [   0    1    3    0  955    0    3    1    2   17]
 [   1    0    0    9    0  876    3    0    2    1]
 [   6    4    0    0    3    6  934    0    5    0]
 [   2    4   18    3    1    0    0  985    2   13]
 [   4    0    4    3    4    1    1    3  950    4]
 [   6    6    0    7    4    5    0    4    3  974]]
```

干净图像上的性能

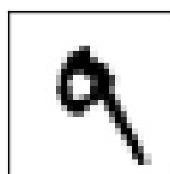
现在将对抗噪声重置为零，看看神经网络在干净图像上的表现。

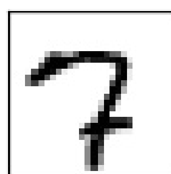
```
init_noise()
```

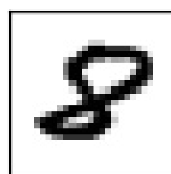
相比噪声图像，神经网络在干净图像上表现的要更差一点。

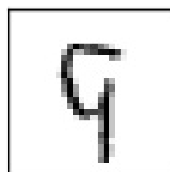
```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

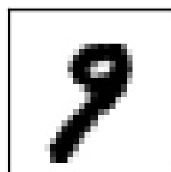
```
Accuracy on Test-Set: 92.2% (9222 / 10000)
Example errors:
```

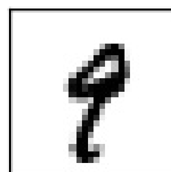

True: 9, Pred: 4

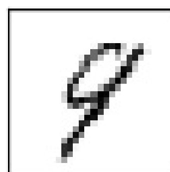

True: 7, Pred: 2

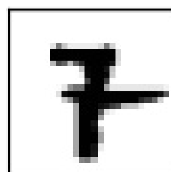

True: 8, Pred: 2

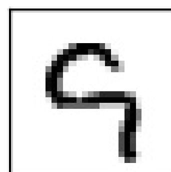

True: 9, Pred: 5


True: 9, Pred: 8


True: 9, Pred: 4


True: 9, Pred: 4


True: 7, Pred: 2


True: 9, Pred: 5

```
Confusion Matrix:
[[ 970    0    1    0    0    1    8    0    0    0]
 [   0 1121    5    0    0    0    9    0    0    0]
 [   2    1 1028    0    0    0    1    0    0    0]
 [   1    0   27  964    0   13    2    2    1    0]
 [   0    2    3    0  957    0   20    0    0    0]
 [   3    0    2    2    0  875   10    0    0    0]
 [   4    1    0    0    1    1  951    0    0    0]
 [  10   21   61    3   14    3    0  913    3    0]
 [  29    2   91    7    7   26   70    1  741    0]
 [  20   18   10   12  150   65   11   12    9  702]]
```

关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。

```
# This has been commented out in case you want to modify and exp
eriment
# with the Notebook without having to restart it.
# session.close()
```

讨论

在上面的实验中可以看到，我们能够使神经网络对单个目标类别的对抗噪声免疫。这使得不可能找到引起误分类到目标类型的对抗噪声。但是，显然也不可能使神经网络同时对所有目标类别免疫。可能用其他方法能够做到这一点。

一种建议是对不同目标类型进行交叉的免疫训练，而不是依次对每个目标类型进行完全的优化。对上面的代码做些小修改就能做到这一点。

另一个建议是设置两层神经网络，共11个网络。第一层网络用来对输入图像进行分类。这个网络没有对对抗噪声免疫。然后根据第一层的预测类型选择第二层的另一个网络。第二层中的网络对各自目标类型的对抗噪声免疫。因此，一个对抗样本可能糊弄第一层的网络，但第二层中的网络会免于特定目标类型噪声的影响。

这可能使用了类型数量比较少的情况，但如果数量很大就变得不可行，比如ImageNet有1000个类别，这样我们在第二层中需要训练1000个神经网络，这并不实际。

总结

这篇教程展示了如何找到MNIST数据集手写数字的对抗噪声。 每个目标类别都找到了一个单一的噪声模式，它导致几乎所有的输入图像都被误分类为目标类别。

MNIST数据集的噪声模式对人眼清晰可见。但可能在高分辨率图像上（比如ImageNet数据集）工作的大型神经网络可以找到更细微的噪声模式。

本教程也尝试了使神经网络免受对抗噪声影响的方法。 这对单个目标类别有效，但所测试的方法无法使神经网络同时对所有对抗目标类别免疫。

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 尝试为对抗噪声使用更少或更多的优化迭代数。
- 教程#11只需少于30次的迭代次数就能找到对抗噪声，相比之下为什么这篇教程需要更多迭代？
- 尝试设置不同的 `noise_limit` 和 `noise_l2_weight` 。这会如何影响对抗噪声以及分类准确率？
- 试着为目标类型1寻找对抗噪声。它是否适用于目标类型3？
- 你能找到一个更好的方法，使得神经网络对对抗噪声免疫吗？
- 神经网络是否可以对单个图像产生的对抗噪声免疫，就像教程 #11 中所做的那样？
- 尝试用不同的配置创建另一个神经网络。一个网络上的对抗噪声对另一个网络有效吗？
- 用CIFAR-10数据集代替MNIST。你可以复用教程 #06 中的一些代码。
- 你会如何找到Inception模型和ImageNet数据集的对抗噪声？
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #13

可视化分析

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist/Github

介绍

在之前的一些关于卷积神经网络的教程中，我们展示了卷积滤波权重，比如教程#02和#06。但单从滤波权重上看，不可能确定卷积滤波器能从输入图像中识别出什么。

本教程中，我们会提出一种用于可视化分析神经网络内部工作原理的基本方法。这个方法就是生成最大化神经网络内个体特征的图像。图像用一些随机噪声初始化，然后用给定特征关于输入图像的梯度来逐渐改变（生成的）图像。

可视化分析神经网络的方法也称为 特征最大化（*feature maximization*） 或 激活最大化（*activation maximization*）。

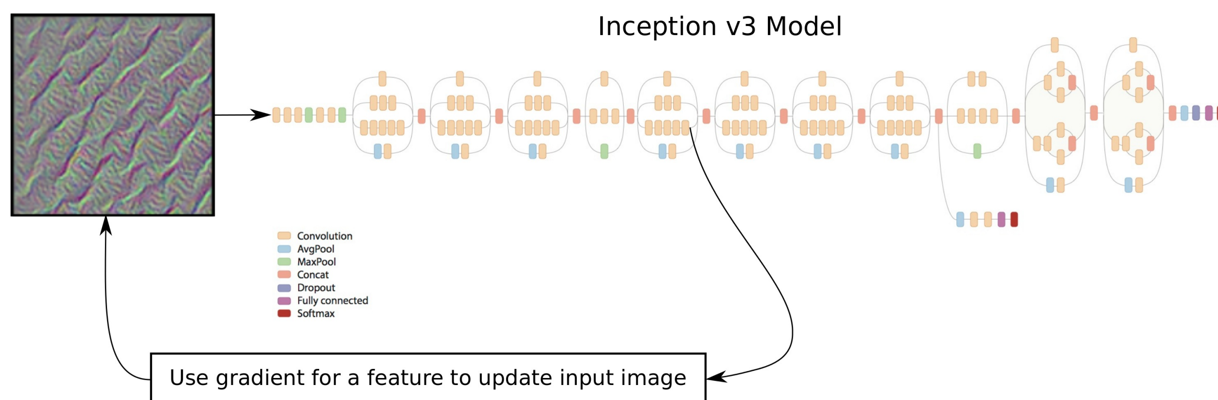
本文基于之前的教程。你需要大概地熟悉神经网络（详见教程 #01和 #02），了解Inception模型也很有帮助（教程 #07）。

流程图

这里将会使用教程 #07中的Inception模型。我们想要找到使得神经网络内给定特征最大化的图像。输入图像用一些噪声初始化，然后用给定特征的梯度来更新图像。在执行了一些优化迭代之后，我们会得到一个这个特定特征“喜欢看到的”图像。

由于Inception模型是由很多相结合的基本数学运算构造的，使用微分链式法则，TensorFlow让我们很快就能找到损失函数的梯度。

```
from IPython.display import Image, display
Image('images/13_visual_analysis_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

# Functions and classes for loading and using the Inception mode
l.
import inception
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'1.1.0'
```

Inception 模型

从网上下载Inception模型

从网上下载Inception模型。这是你保存数据文件的默认文件夹。如果文件夹不存在就自动创建。

```
# inception.data_dir = 'inception/'
```

如果文件夹中不存在Inception模型，就自动下载。 它有85MB。

```
inception.maybe_download()
```

```
Downloading Inception v3 Model ...
- Download progress: 100.0%
Download finished. Extracting files.
Done.
```

卷积层的名称

这个函数返回Inception模型中卷积层的名称列表。

```
def get_conv_layer_names():
    # Load the Inception model.
    model = inception.Inception()

    # Create a list of names for the operations in the graph
    # for the Inception model where the operator-type is 'Conv2D
'.
    names = [op.name for op in model.graph.get_operations() if o
p.type=='Conv2D']

    # Close the TensorFlow session inside the model-object.
    model.close()

    return names
```

```
conv_names = get_conv_layer_names()
```

在Inception模型中总共有94个卷积层。

```
len(conv_names)
```

```
94
```

写出头5个卷积层的名称。

```
conv_names[:5]
```

```
['conv/Conv2D',
 'conv_1/Conv2D',
 'conv_2/Conv2D',
 'conv_3/Conv2D',
 'conv_4/Conv2D']
```

写出最后5个卷积层的名称。

```
conv_names[-5:]
```

```
['mixed_10/tower_1/conv/Conv2D',
 'mixed_10/tower_1/conv_1/Conv2D',
 'mixed_10/tower_1/mixed/conv/Conv2D',
 'mixed_10/tower_1/mixed/conv_1/Conv2D',
 'mixed_10/tower_2/conv/Conv2D']
```

找到输入图像的帮助函数

这个函数用来寻找使网络内给定特征最大化的输入图像。它本质上是用梯度法来进行优化。图像用小的随机值初始化，然后用给定特征关于输入图像的梯度来逐步更新。

```
def optimize_image(conv_id=None, feature=0,
                   num_iterations=30, show_progress=True):
    """
    Find an image that maximizes the feature
    given by the conv_id and feature number.

    Parameters:
    conv_id: Integer identifying the convolutional layer to
             maximize. It is an index into conv_names.
             If None then use the last fully-connected layer
             before the softmax output.
    feature: Index into the layer for the feature to maximize.
    num_iteration: Number of optimization iterations to perform.
    show_progress: Boolean whether to show the progress.
    """

    # Load the Inception model. This is done for each call of
    # this function because we will add a lot to the graph
    # which will cause the graph to grow and eventually the
    # computer will run out of memory.
    model = inception.Inception()

    # Reference to the tensor that takes the raw input image.
    resized_image = model.resized_image
```

```
    # Reference to the tensor for the predicted classes.
    # This is the output of the final layer's softmax classifier.

    y_pred = model.y_pred

    # Create the loss-function that must be maximized.
    if conv_id is None:
        # If we want to maximize a feature on the last layer,
        # then we use the fully-connected layer prior to the
        # softmax-classifier. The feature no. is the class-number

        # and must be an integer between 1 and 1000.
        # The loss-function is just the value of that feature.
        loss = model.y_logits[0, feature]
    else:
        # If instead we want to maximize a feature of a
        # convolutional layer inside the neural network.

        # Get the name of the convolutional operator.
        conv_name = conv_names[conv_id]

        # Get a reference to the tensor that is output by the
        # operator. Note that ":0" is added to the name for this.

        tensor = model.graph.get_tensor_by_name(conv_name + ":0"
)

        # Set the Inception model's graph as the default
        # so we can add an operator to it.
        with model.graph.as_default():
            # The loss-function is the average of all the
            # tensor-values for the given feature. This
            # ensures that we generate the whole input image.
            # You can try and modify this so it only uses
            # a part of the tensor.
            loss = tf.reduce_mean(tensor[:,:,:,feature])

    # Get the gradient for the loss-function with regard to
    # the resized input image. This creates a mathematical
    # function for calculating the gradient.
    gradient = tf.gradients(loss, resized_image)

    # Create a TensorFlow session so we can run the graph.
    session = tf.Session(graph=model.graph)

    # Generate a random image of the same size as the raw input.
    # Each pixel is a small random value between 128 and 129,
    # which is about the middle of the colour-range.
    image_shape = resized_image.get_shape()
    image = np.random.uniform(size=image_shape) + 128.0

    # Perform a number of optimization iterations to find
```

```
    # the image that maximizes the loss-function.
    for i in range(num_iterations):
        # Create a feed-dict. This feeds the image to the
        # tensor in the graph that holds the resized image, beca
use
        # this is the final stage for inputting raw image data.
        feed_dict = {model.tensor_name_resized_image: image}

        # Calculate the predicted class-scores,
        # as well as the gradient and the loss-value.
        pred, grad, loss_value = session.run([y_pred, gradient,
loss],
                                             feed_dict=feed_dict
)

        # Squeeze the dimensionality for the gradient-array.
        grad = np.array(grad).squeeze()

        # The gradient now tells us how much we need to change t
he
        # input image in order to maximize the given feature.

        # Calculate the step-size for updating the image.
        # This step-size was found to give fast convergence.
        # The addition of 1e-8 is to protect from div-by-zero.
        step_size = 1.0 / (grad.std() + 1e-8)

        # Update the image by adding the scaled gradient
        # This is called gradient ascent.
        image += step_size * grad

        # Ensure all pixel-values in the image are between 0 and
 255.
        image = np.clip(image, 0.0, 255.0)

        if show_progress:
            print("Iteration:", i)

            # Convert the predicted class-scores to a one-dim ar
ray.
            pred = np.squeeze(pred)

            # The predicted class for the Inception model.
            pred_cls = np.argmax(pred)

            # Name of the predicted class.
            cls_name = model.name_lookup.cls_to_name(pred_cls,
                                                     only_first_name=T
rue)

            # The score (probability) for the predicted class.
            cls_score = pred[pred_cls]
```

```
            # Print the predicted score etc.
            msg = "Predicted class-name: {0} (#{1}), score: {2:>
7.2%}"
            print(msg.format(cls_name, pred_cls, cls_score))

            # Print statistics for the gradient.
            msg = "Gradient min: {0:>9.6f}, max: {1:>9.6f}, step
size: {2:>9.2f}"
            print(msg.format(grad.min(), grad.max(), step_size))

            # Print the loss-value.
            print("Loss:", loss_value)

            # Newline.
            print()

    # Close the TensorFlow session inside the model-object.
    model.close()

    return image.squeeze()
```

绘制图像和噪声的帮助函数

函数对图像做归一化，则像素值在0.0到1.0之间。

```
def normalize_image(x):
    # Get the min and max values for all pixels in the input.
    x_min = x.min()
    x_max = x.max()

    # Normalize so all values are between 0.0 and 1.0
    x_norm = (x - x_min) / (x_max - x_min)

    return x_norm
```

这个函数绘制一张图像。

```
def plot_image(image):
    # Normalize the image so pixels are between 0.0 and 1.0
    img_norm = normalize_image(image)

    # Plot the image.
    plt.imshow(img_norm, interpolation='nearest')
    plt.show()
```

这个函数在坐标系内绘制6张图。

```
def plot_images(images, show_size=100):
    """
    The show_size is the number of pixels to show for each image
.
    The max value is 299.
    """

    # Create figure with sub-plots.
    fig, axes = plt.subplots(2, 3)

    # Adjust vertical spacing.
    fig.subplots_adjust(hspace=0.1, wspace=0.1)

    # Use interpolation to smooth pixels?
    smooth = True

    # Interpolation type.
    if smooth:
        interpolation = 'spline16'
    else:
        interpolation = 'nearest'

    # For each entry in the grid.
    for i, ax in enumerate(axes.flat):
        # Get the i'th image and only use the desired pixels.
        img = images[i, 0:show_size, 0:show_size, :]

        # Normalize the image so its pixels are between 0.0 and
1.0
        img_norm = normalize_image(img)

        # Plot the image.
        ax.imshow(img_norm, interpolation=interpolation)

        # Remove ticks.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

优化和绘制图像的帮助函数

这个函数优化多张图像并绘制它们。

```
def optimize_images(conv_id=None, num_iterations=30, show_size=1
00):
    """
    Find 6 images that maximize the 6 first features in the laye
```

```
r
    given by the conv_id.

    Parameters:
    conv_id: Integer identifying the convolutional layer to
             maximize. It is an index into conv_names.
             If None then use the last layer before the softmax
output.
    num_iterations: Number of optimization iterations to perform
.
    show_size: Number of pixels to show for each image. Max 299.
    """

    # Which layer are we using?
    if conv_id is None:
        print("Final fully-connected layer before softmax.")
    else:
        print("Layer:", conv_names[conv_id])

    # Initialize the array of images.
    images = []

    # For each feature do the following. Note that the
    # last fully-connected layer only supports numbers
    # between 1 and 1000, while the convolutional layers
    # support numbers between 0 and some other number.
    # So we just use the numbers between 1 and 7.
    for feature in range(1,7):
        print("Optimizing image for feature no.", feature)

        # Find the image that maximizes the given feature
        # for the network layer identified by conv_id (or None).
        image = optimize_image(conv_id=conv_id, feature=feature,
                               show_progress=False,
                               num_iterations=num_iterations)

        # Squeeze the dim of the array.
        image = image.squeeze()

        # Append to the list of images.
        images.append(image)

    # Convert to numpy-array so we can index all dimensions easi
ly.
    images = np.array(images)

    # Plot the images.
    plot_images(images=images, show_size=show_size)
```

结果

为浅处的卷积层优化图像

举个例子，寻找让卷积层 `conv_names[conv_id]` 中的2号特征最大化的输入图像，其中 `conv_id=5` 。

```
image = optimize_image(conv_id=5, feature=2,
                       num_iterations=30, show_progress=True)
```

```
Iteration: 0
Predicted class-name: dishwasher (#667), score:   4.81%
Gradient min: -0.000083, max:  0.000100, stepsize:  76290.32
Loss: 4.83793

Iteration: 1
Predicted class-name: kite (#397), score:  15.12%
Gradient min: -0.000142, max:  0.000126, stepsize:  71463.42
Loss: 5.59611

Iteration: 2
Predicted class-name: wall clock (#524), score:   6.85%
Gradient min: -0.000119, max:  0.000121, stepsize:  80427.39
Loss: 6.91725

Iteration: 3
Predicted class-name: syringe (#531), score:   4.69%
Gradient min: -0.000124, max:  0.000116, stepsize:  87046.41
Loss: 7.93267

Iteration: 4
Predicted class-name: syringe (#531), score:   6.53%
Gradient min: -0.000115, max:  0.000122, stepsize:  94634.06
Loss: 8.85594

Iteration: 5
Predicted class-name: syringe (#531), score:  21.31%
Gradient min: -0.000108, max:  0.000131, stepsize: 103182.49
Loss: 9.70698

Iteration: 6
Predicted class-name: syringe (#531), score:  36.39%
Gradient min: -0.000102, max:  0.000099, stepsize: 111440.73
Loss: 10.4558

Iteration: 7
Predicted class-name: syringe (#531), score:  43.79%
Gradient min: -0.000100, max:  0.000083, stepsize: 119285.09
Loss: 11.1371

Iteration: 8
Predicted class-name: syringe (#531), score:  34.85%
```

```
Gradient min: -0.000078, max:  0.000098, stepsize: 126258.06
Loss: 11.7331

Iteration: 9
Predicted class-name: syringe (#531), score:  18.28%
Gradient min: -0.000075, max:  0.000071, stepsize: 133766.53
Loss: 12.2777

Iteration: 10
Predicted class-name: syringe (#531), score:  11.91%
Gradient min: -0.000072, max:  0.000079, stepsize: 139181.44
Loss: 12.7673

Iteration: 11
Predicted class-name: binder (#835), score:  13.27%
Gradient min: -0.000079, max:  0.000070, stepsize: 145263.47
Loss: 13.2062

Iteration: 12
Predicted class-name: binder (#835), score:  15.05%
Gradient min: -0.000060, max:  0.000101, stepsize: 150589.72
Loss: 13.6149

Iteration: 13
Predicted class-name: binder (#835), score:  14.79%
Gradient min: -0.000074, max:  0.000072, stepsize: 156626.62
Loss: 13.9922

Iteration: 14
Predicted class-name: binder (#835), score:  14.44%
Gradient min: -0.000078, max:  0.000062, stepsize: 160979.04
Loss: 14.3428

Iteration: 15
Predicted class-name: binder (#835), score:  11.76%
Gradient min: -0.000081, max:  0.000081, stepsize: 164249.60
Loss: 14.6689

Iteration: 16
Predicted class-name: binder (#835), score:   9.61%
Gradient min: -0.000069, max:  0.000073, stepsize: 169375.77
Loss: 14.968

Iteration: 17
Predicted class-name: binder (#835), score:   7.51%
Gradient min: -0.000060, max:  0.000086, stepsize: 173951.43
Loss: 15.2644

Iteration: 18
Predicted class-name: binder (#835), score:   6.16%
Gradient min: -0.000057, max:  0.000074, stepsize: 176921.49
Loss: 15.5303
```

```
Iteration: 19
Predicted class-name: quilt (#976), score:   6.22%
Gradient min: -0.000067, max:  0.000068, stepsize: 182788.52
Loss: 15.7967

Iteration: 20
Predicted class-name: quilt (#976), score:   7.31%
Gradient min: -0.000068, max:  0.000063, stepsize: 185266.16
Loss: 16.0442

Iteration: 21
Predicted class-name: bib (#941), score:   7.74%
Gradient min: -0.000048, max:  0.000066, stepsize: 190195.76
Loss: 16.2883

Iteration: 22
Predicted class-name: bib (#941), score:   9.43%
Gradient min: -0.000060, max:  0.000047, stepsize: 192709.62
Loss: 16.5165

Iteration: 23
Predicted class-name: bib (#941), score:  11.05%
Gradient min: -0.000064, max:  0.000049, stepsize: 197288.09
Loss: 16.7361

Iteration: 24
Predicted class-name: bib (#941), score:  12.59%
Gradient min: -0.000054, max:  0.000047, stepsize: 201010.69
Loss: 16.9544

Iteration: 25
Predicted class-name: bib (#941), score:  15.13%
Gradient min: -0.000045, max:  0.000049, stepsize: 204798.67
Loss: 17.1659

Iteration: 26
Predicted class-name: bib (#941), score:  15.91%
Gradient min: -0.000047, max:  0.000047, stepsize: 208499.70
Loss: 17.3637

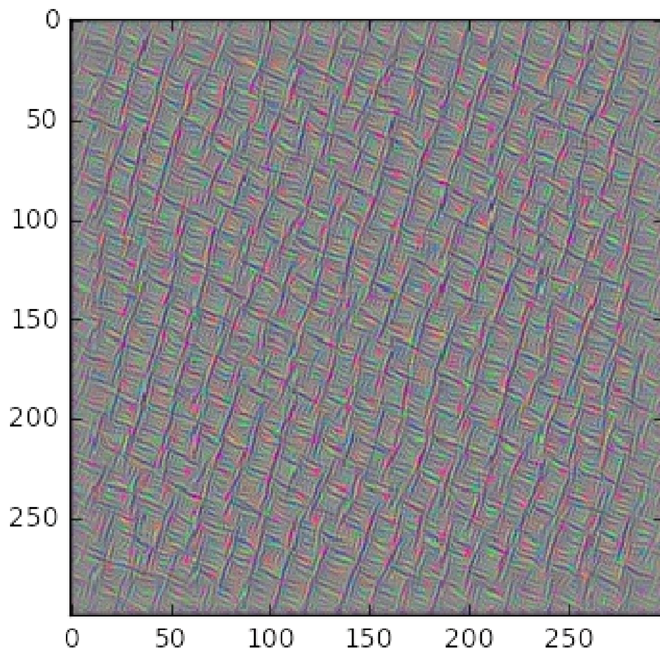
Iteration: 27
Predicted class-name: bib (#941), score:  17.96%
Gradient min: -0.000056, max:  0.000059, stepsize: 210286.13
Loss: 17.559

Iteration: 28
Predicted class-name: bib (#941), score:  19.26%
Gradient min: -0.000043, max:  0.000043, stepsize: 214742.82
Loss: 17.7469

Iteration: 29
Predicted class-name: bib (#941), score:  18.87%
Gradient min: -0.000047, max:  0.000059, stepsize: 218511.00
```

```
Loss: 17.9321
```

```
plot_image(image)
```



为卷积层优化多张图像

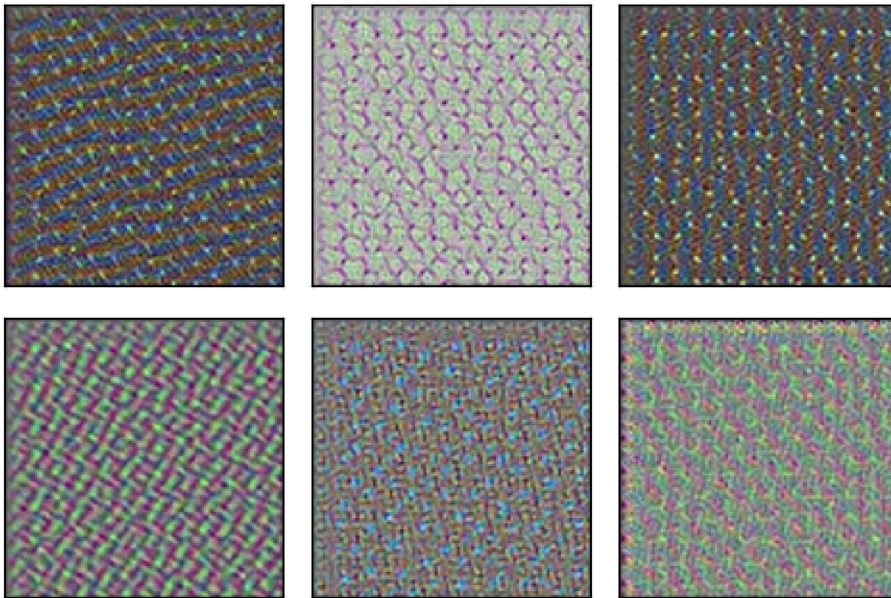
下面，我们为Inception模型中的卷积层优化多张图像，并绘制它们。这些图像展示了卷积层“想看到的”内容。注意更深的层次里图案变得越来越复杂。

```
optimize_images(conv_id=0, num_iterations=10)
```

```
Layer: conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
```

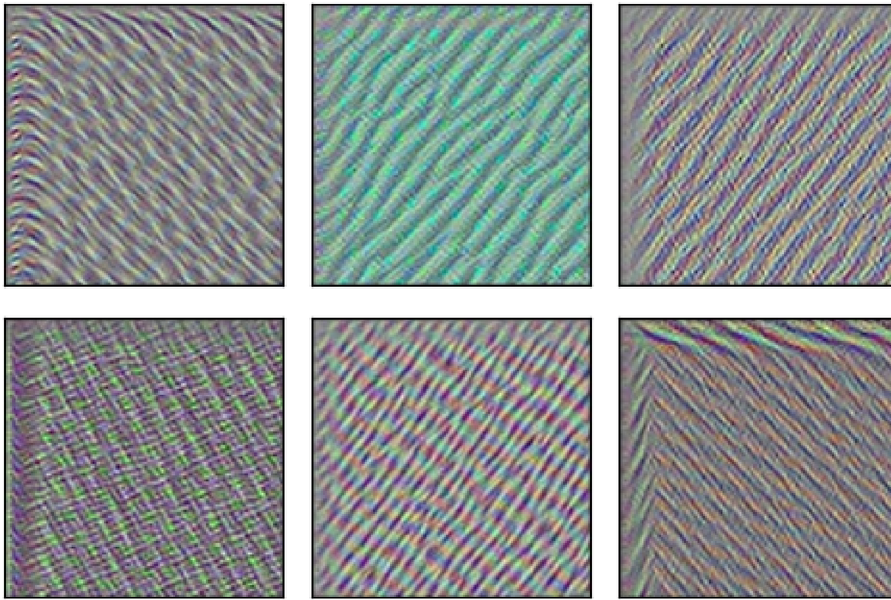
```
optimize_images(conv_id=3, num_iterations=30)
```

```
Layer: conv_3/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



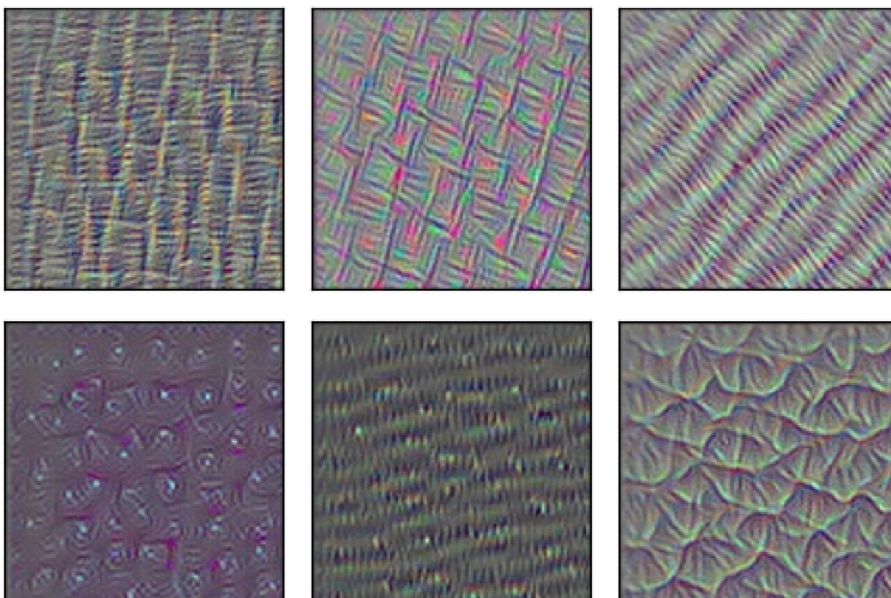
```
optimize_images(conv_id=4, num_iterations=30)
```

```
Layer: conv_4/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



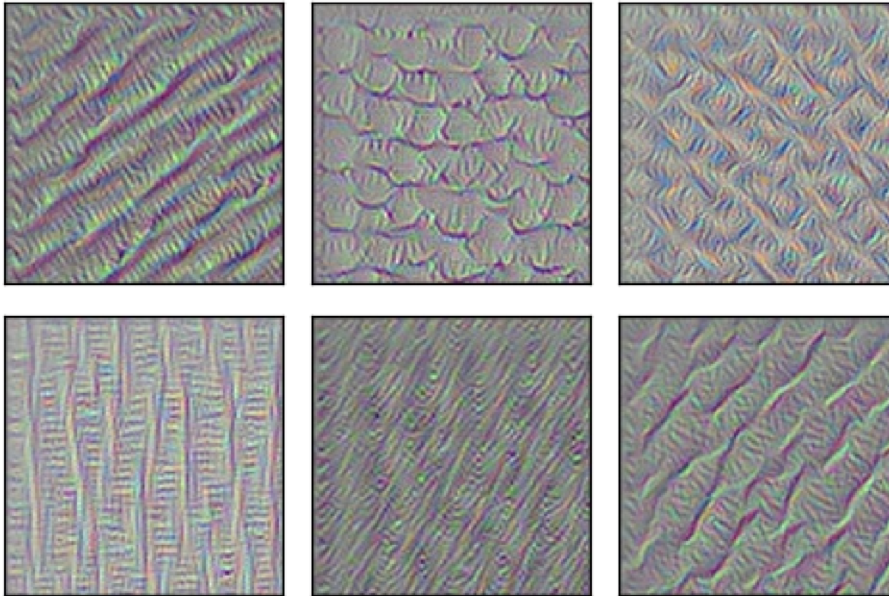
```
optimize_images(conv_id=5, num_iterations=30)
```

```
Layer: mixed/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



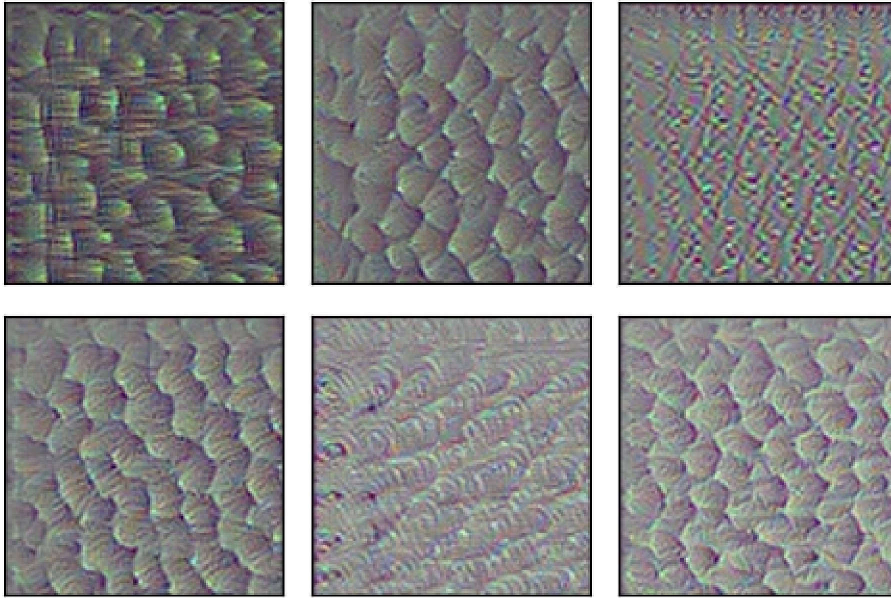
```
optimize_images(conv_id=6, num_iterations=30)
```

```
Layer: mixed/tower/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



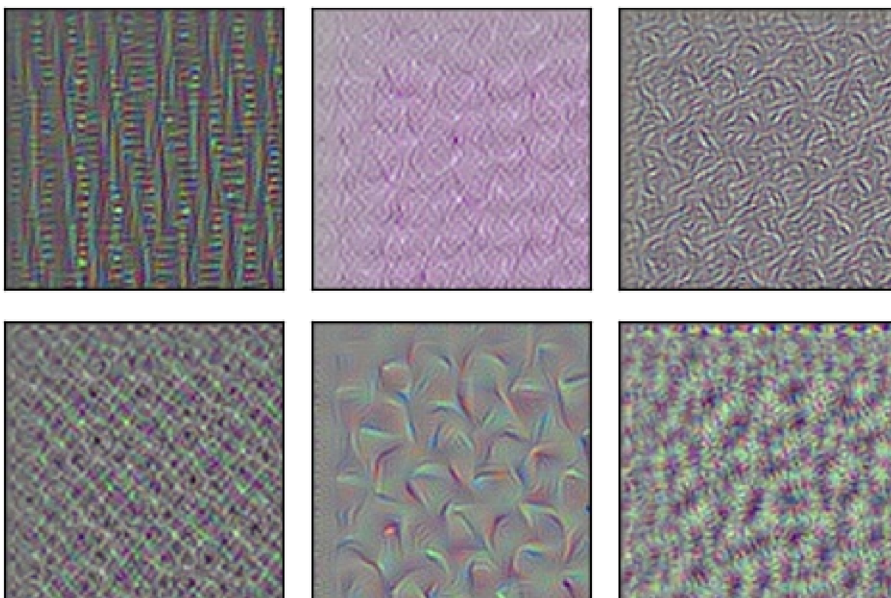
```
optimize_images(conv_id=7, num_iterations=30)
```

```
Layer: mixed/tower/conv_1/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



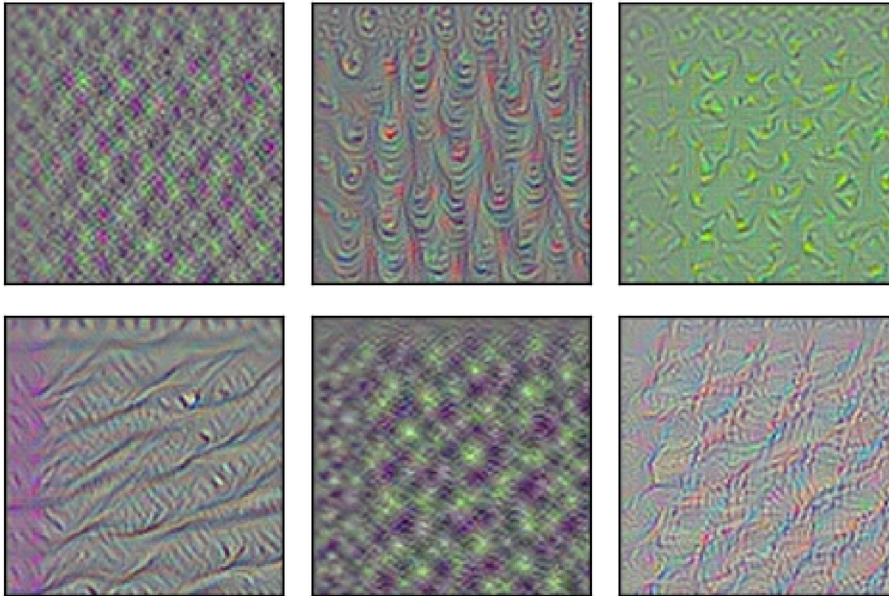
```
optimize_images(conv_id=8, num_iterations=30)
```

```
Layer: mixed/tower_1/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



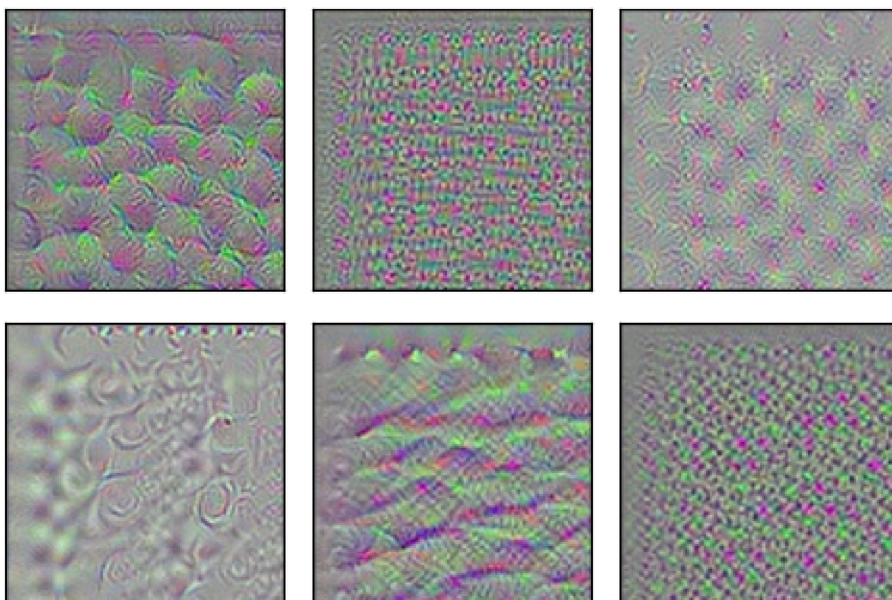
```
optimize_images(conv_id=9, num_iterations=30)
```

```
Layer: mixed/tower_1/conv_1/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



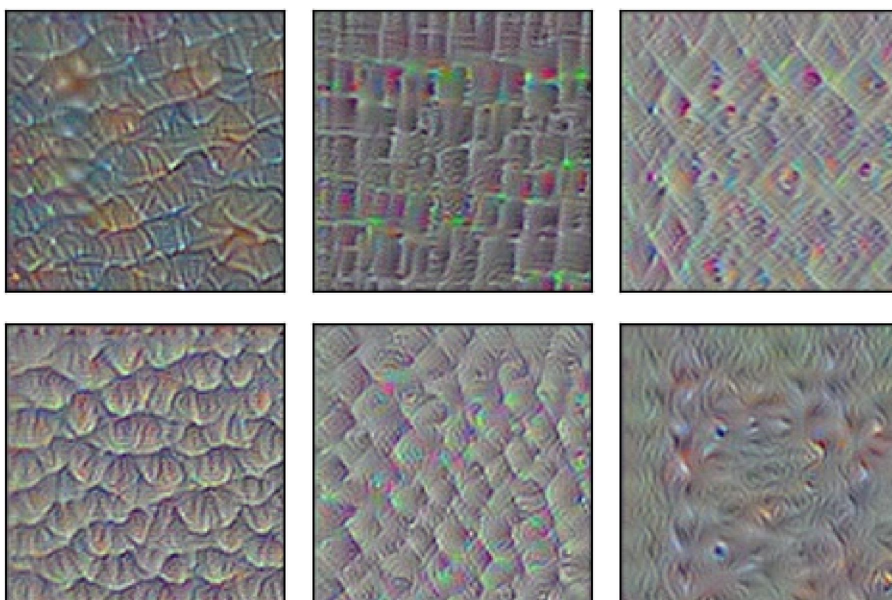
```
optimize_images(conv_id=10, num_iterations=30)
```

```
Layer: mixed/tower_1/conv_2/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



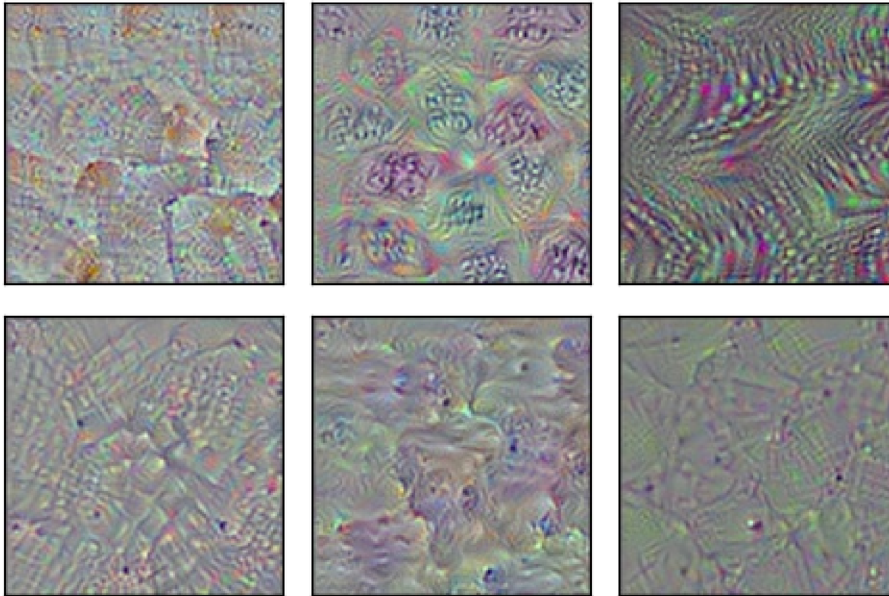
```
optimize_images(conv_id=20, num_iterations=30)
```

```
Layer: mixed_2/tower/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



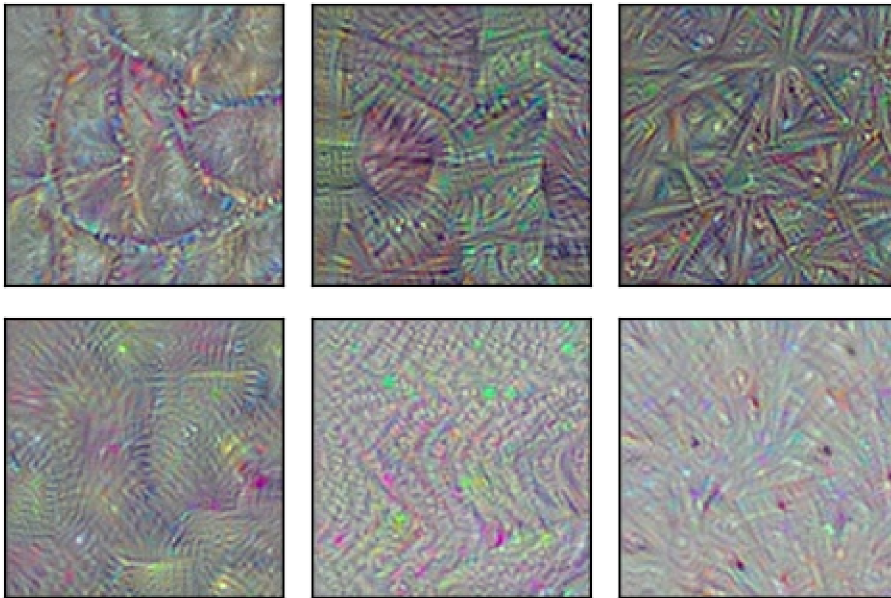
```
optimize_images(conv_id=30, num_iterations=30)
```

```
Layer: mixed_4/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



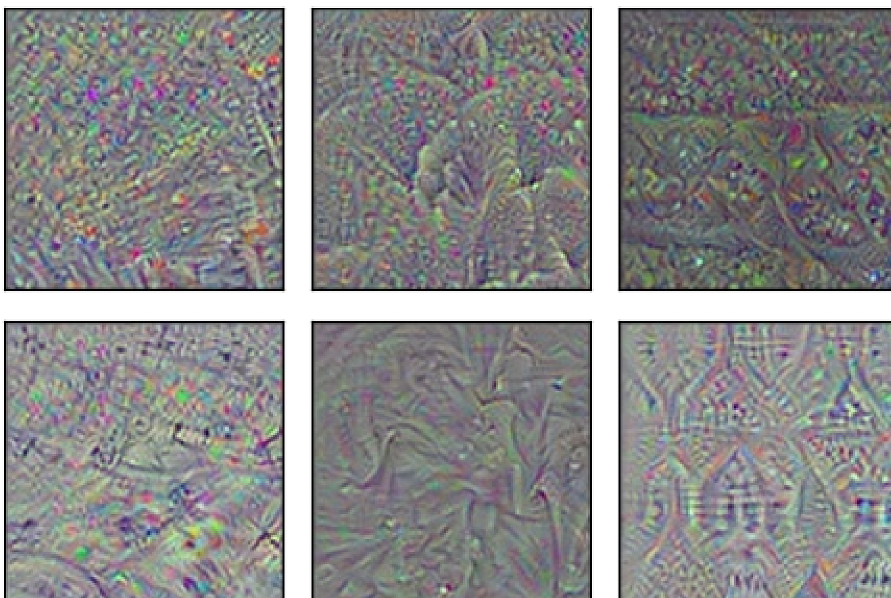
```
optimize_images(conv_id=40, num_iterations=30)
```

```
Layer: mixed_5/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



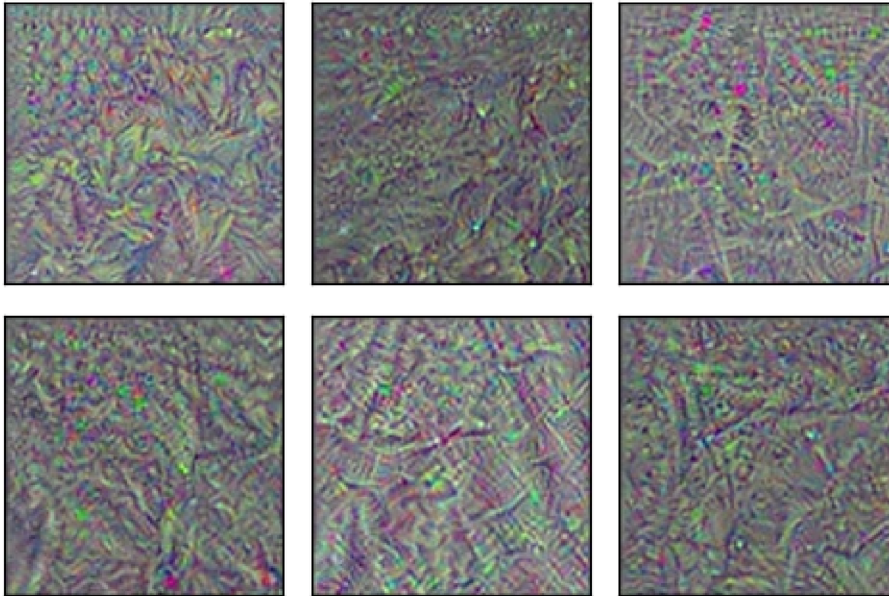
```
optimize_images(conv_id=50, num_iterations=30)
```

```
Layer: mixed_6/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



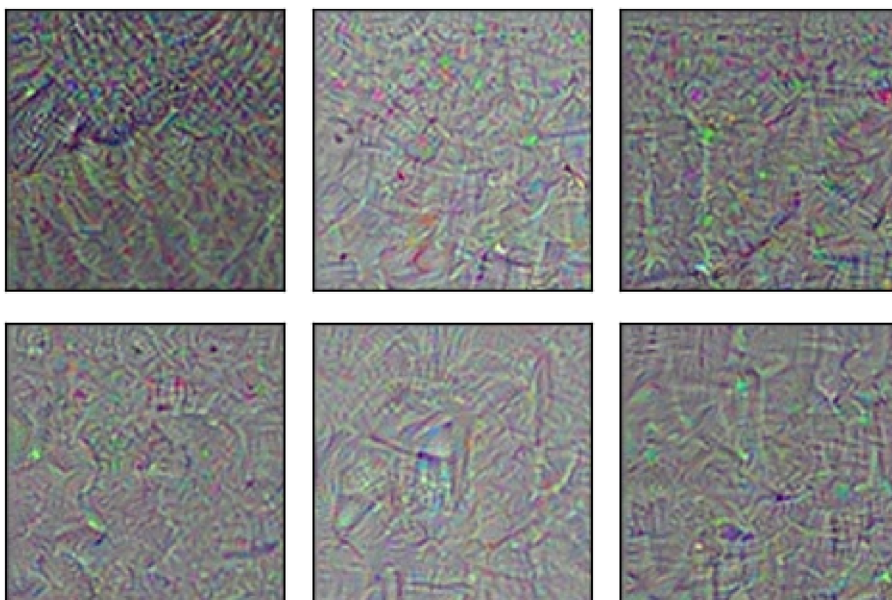
```
optimize_images(conv_id=60, num_iterations=30)
```

```
Layer: mixed_7/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



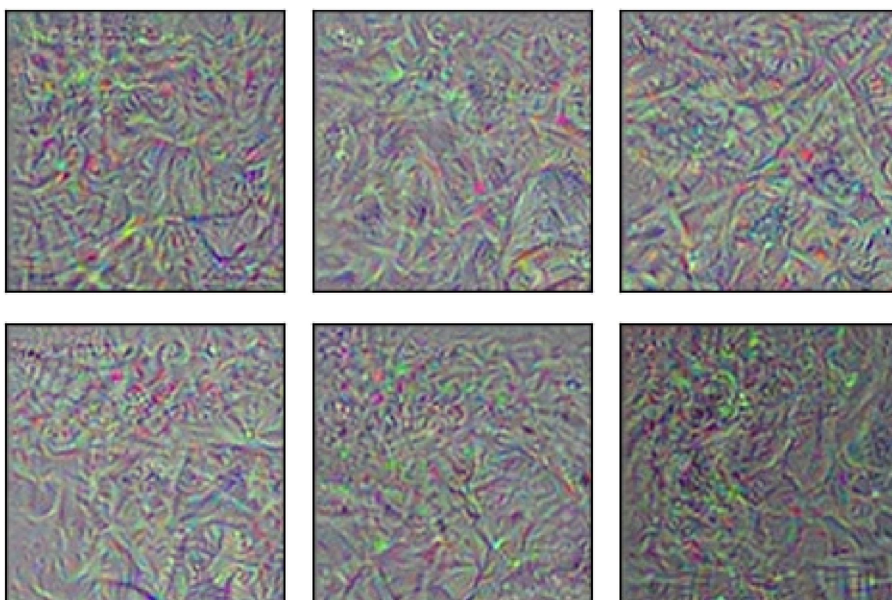
```
optimize_images(conv_id=70, num_iterations=30)
```

```
Layer: mixed_8/tower/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



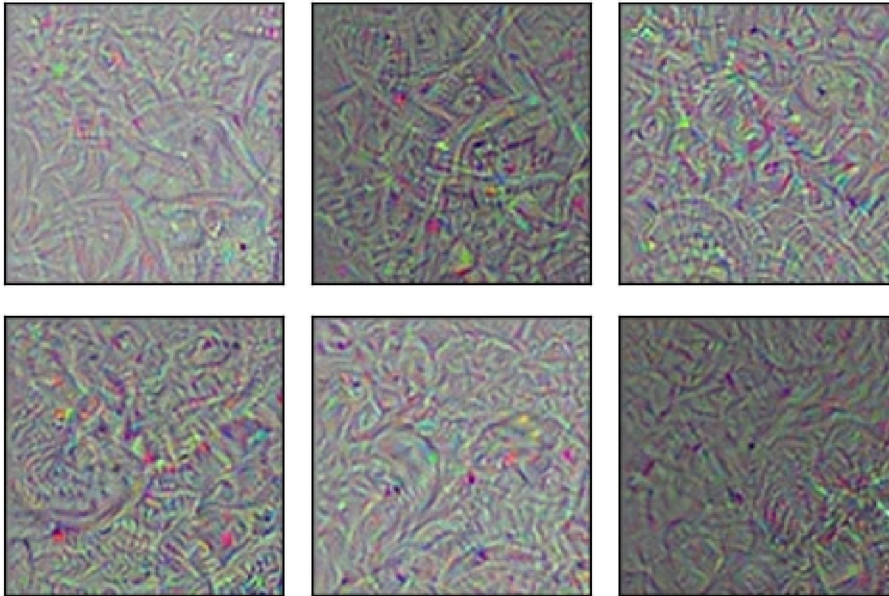
```
optimize_images(conv_id=80, num_iterations=30)
```

```
Layer: mixed_9/tower_1/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



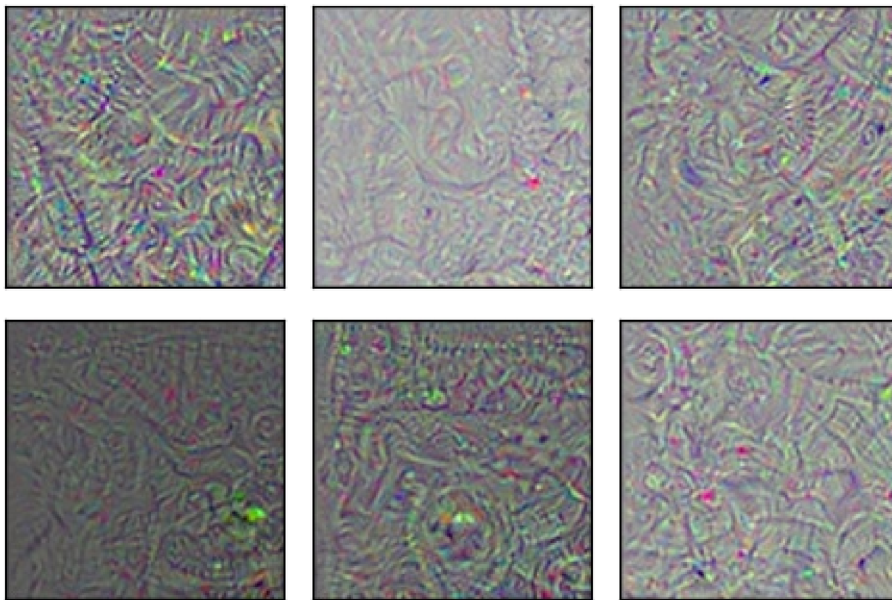
```
optimize_images(conv_id=90, num_iterations=30)
```

```
Layer: mixed_10/tower_1/conv_1/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



```
optimize_images(conv_id=93, num_iterations=30)
```

```
Layer: mixed_10/tower_2/conv/Conv2D
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



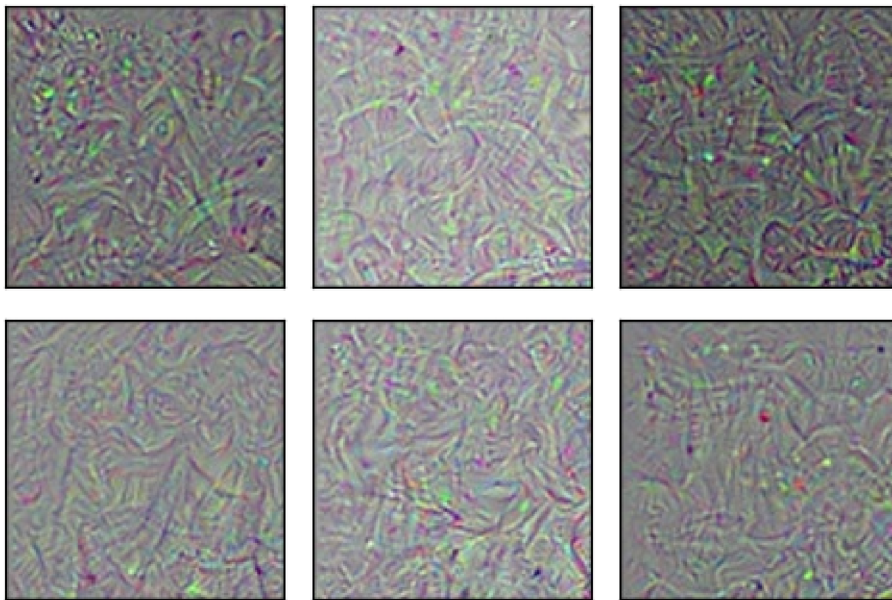
Softmax前最终的全连接层

现在，我们为Inception模型中的最后一层优化并绘制图像。这是在softmax分类器前的全连接层。该层特征对应了输出的类别。

我们可能希望在这些图像里看到一些可识别的图案，比如对应输出类别的猴子、鸟类等，但图像只显示了一些复杂的、抽象的图案。

```
optimize_images(conv_id=None, num_iterations=30)
```

```
Final fully-connected layer before softmax.
Optimizing image for feature no. 1
Optimizing image for feature no. 2
Optimizing image for feature no. 3
Optimizing image for feature no. 4
Optimizing image for feature no. 5
Optimizing image for feature no. 6
```



上面只显示了100x100像素的图像，但实际上是299x299像素。如果我们执行更多的优化迭代并画出完整的图像，可能会有一些可识别的模式。那么，让我们再次优化第一张图像，并以全分辨率来绘制。

Inception模型以大约100%的确信度将结果图像分类成“敏狐”，但在人眼看来，图像只是一些抽象的图案。

如果你想测试另一个特征号码，要注意，号码必须介于0到1000之间，因为它对应了最终输出层的一个有效类别号。

```
image = optimize_image(conv_id=None, feature=1,
                       num_iterations=100, show_progress=True)
```

```
Iteration: 0
Predicted class-name: dishwasher (#667), score:   4.98%
Gradient min: -0.006252, max:  0.004451, stepsize:   3734.48
Loss: -0.837608

Iteration: 1
Predicted class-name: ballpoint (#907), score:   8.52%
Gradient min: -0.007303, max:  0.006427, stepsize:   2152.89
Loss: -0.416723

Iteration: 2
Predicted class-name: spider web (#600), score:  90.44%
Gradient min: -0.007480, max:  0.012272, stepsize:   1343.66
Loss: 2.77814

Iteration: 3
Predicted class-name: pot (#838), score:   3.01%
Gradient min: -0.009853, max:  0.007638, stepsize:   1526.98
Loss: 3.27751
```

```
Iteration: 4
Predicted class-name: American egret (#426), score:  11.55%
Gradient min: -0.008507, max:  0.006308, stepsize:   1787.96
Loss: 5.95497

Iteration: 5
Predicted class-name: spider web (#600), score:  21.98%
Gradient min: -0.010415, max:  0.009410, stepsize:   1722.27
Loss: 5.07394

Iteration: 6
Predicted class-name: kit fox (#1), score:  29.21%
Gradient min: -0.009298, max:  0.007885, stepsize:   2471.91
Loss: 7.98241

Iteration: 7
Predicted class-name: brain coral (#649), score:   8.95%
Gradient min: -0.004683, max:  0.004366, stepsize:   2876.78
Loss: 2.61856

Iteration: 8
Predicted class-name: kit fox (#1), score:  30.05%
Gradient min: -0.008918, max:  0.006374, stepsize:   2243.47
Loss: 8.18703

Iteration: 9
Predicted class-name: kit fox (#1), score:  55.10%
Gradient min: -0.041977, max:  0.025564, stepsize:   1270.96
Loss: 9.4695

Iteration: 10
Predicted class-name: kit fox (#1), score:  45.79%
Gradient min: -0.025474, max:  0.026726, stepsize:    858.88
Loss: 8.49094

Iteration: 11
Predicted class-name: kit fox (#1), score:  71.39%
Gradient min: -0.021939, max:  0.016643, stepsize:   1316.54
Loss: 12.4999

Iteration: 12
Predicted class-name: kit fox (#1), score:  82.78%
Gradient min: -0.011797, max:  0.017714, stepsize:   1763.08
Loss: 11.1421

Iteration: 13
Predicted class-name: kit fox (#1), score:  67.38%
Gradient min: -0.016686, max:  0.015832, stepsize:   1908.94
Loss: 11.186

Iteration: 14
Predicted class-name: kit fox (#1), score:  57.64%
```

```
Gradient min: -0.017312, max:  0.014563, stepsize:   1412.06
Loss: 9.67373

Iteration: 15
Predicted class-name: kit fox (#1), score:  69.09%
Gradient min: -0.005773, max:  0.005870, stepsize:   3163.64
Loss: 12.9428

Iteration: 16
Predicted class-name: kit fox (#1), score:  82.98%
Gradient min: -0.020765, max:  0.017950, stepsize:   1225.38
Loss: 10.6559

Iteration: 17
Predicted class-name: kit fox (#1), score:  99.04%
Gradient min: -0.005492, max:  0.006126, stepsize:   2520.72
Loss: 16.579

Iteration: 18
Predicted class-name: kit fox (#1), score:  86.78%
Gradient min: -0.017253, max:  0.028574, stepsize:   1280.11
Loss: 11.7084

Iteration: 19
Predicted class-name: kit fox (#1), score:  96.57%
Gradient min: -0.007056, max:  0.006660, stepsize:   1838.04
Loss: 17.8698

Iteration: 20
Predicted class-name: kit fox (#1), score:  99.04%
Gradient min: -0.008916, max:  0.008408, stepsize:   2720.73
Loss: 17.922

Iteration: 21
Predicted class-name: kit fox (#1), score:  68.39%
Gradient min: -0.012104, max:  0.013627, stepsize:   1398.73
Loss: 12.5718

Iteration: 22
Predicted class-name: kit fox (#1), score:  98.38%
Gradient min: -0.007660, max:  0.007840, stepsize:   2043.20
Loss: 14.0164

Iteration: 23
Predicted class-name: kit fox (#1), score:  98.36%
Gradient min: -0.009233, max:  0.006748, stepsize:   1951.74
Loss: 19.118

Iteration: 24
Predicted class-name: kit fox (#1), score:  99.44%
Gradient min: -0.013526, max:  0.015166, stepsize:   1557.67
Loss: 23.2171
```

```
Iteration: 25
Predicted class-name: kit fox (#1), score:  99.83%
Gradient min: -0.005306, max:  0.006063, stepsize:   2142.04
Loss: 21.0666

Iteration: 26
Predicted class-name: kit fox (#1), score:  99.85%
Gradient min: -0.005931, max:  0.005094, stepsize:   2287.80
Loss: 21.2772

Iteration: 27
Predicted class-name: kit fox (#1), score:  97.91%
Gradient min: -0.008425, max:  0.010999, stepsize:   1633.57
Loss: 23.1276

Iteration: 28
Predicted class-name: kit fox (#1), score:  99.98%
Gradient min: -0.012720, max:  0.010505, stepsize:   1749.55
Loss: 25.9384

Iteration: 29
Predicted class-name: kit fox (#1), score:  99.90%
Gradient min: -0.020819, max:  0.023275, stepsize:   1026.48
Loss: 22.4687

Iteration: 30
Predicted class-name: kit fox (#1), score:  99.87%
Gradient min: -0.005569, max:  0.007158, stepsize:   2436.42
Loss: 21.3727

Iteration: 31
Predicted class-name: kit fox (#1), score:  97.25%
Gradient min: -0.010902, max:  0.007087, stepsize:   1689.47
Loss: 13.2659

Iteration: 32
Predicted class-name: kit fox (#1), score:  99.96%
Gradient min: -0.006695, max:  0.006514, stepsize:   2277.89
Loss: 23.113

Iteration: 33
Predicted class-name: kit fox (#1), score:  99.89%
Gradient min: -0.011343, max:  0.011963, stepsize:   1713.67
Loss: 20.4645

Iteration: 34
Predicted class-name: kit fox (#1), score:  99.83%
Gradient min: -0.005129, max:  0.005226, stepsize:   2531.23
Loss: 22.9016

Iteration: 35
Predicted class-name: kit fox (#1), score:  99.96%
Gradient min: -0.004618, max:  0.005916, stepsize:   1979.85
```

```
Loss: 19.6406

Iteration: 36
Predicted class-name: kit fox (#1), score:  99.94%
Gradient min: -0.005298, max:  0.007882, stepsize:   2158.99
Loss: 26.6898

Iteration: 37
Predicted class-name: kit fox (#1), score:  99.77%
Gradient min: -0.009913, max:  0.010110, stepsize:   1643.65
Loss: 21.2908

Iteration: 38
Predicted class-name: kit fox (#1), score:  99.99%
Gradient min: -0.005472, max:  0.004434, stepsize:   2654.99
Loss: 28.4096

Iteration: 39
Predicted class-name: kit fox (#1), score:  99.99%
Gradient min: -0.006044, max:  0.007171, stepsize:   1646.69
Loss: 32.005

Iteration: 40
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.007782, max:  0.007306, stepsize:   1853.00
Loss: 34.7635

Iteration: 41
Predicted class-name: kit fox (#1), score:  99.94%
Gradient min: -0.030789, max:  0.017443, stepsize:   1224.15
Loss: 32.2997

Iteration: 42
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005752, max:  0.006784, stepsize:   2148.87
Loss: 34.8329

Iteration: 43
Predicted class-name: kit fox (#1), score:  99.98%
Gradient min: -0.005747, max:  0.005908, stepsize:   2058.32
Loss: 33.3857

Iteration: 44
Predicted class-name: kit fox (#1), score:  99.99%
Gradient min: -0.005644, max:  0.005296, stepsize:   2042.49
Loss: 33.5334

Iteration: 45
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.008353, max:  0.008290, stepsize:   1814.68
Loss: 34.6049

Iteration: 46
```

```
Predicted class-name: kit fox (#1), score:  99.99%
Gradient min: -0.007643, max:  0.006041, stepsize:   2002.25
Loss: 36.0055

Iteration: 47
Predicted class-name: kit fox (#1), score:  99.99%
Gradient min: -0.008912, max:  0.009060, stepsize:   1462.12
Loss: 31.0812

Iteration: 48
Predicted class-name: kit fox (#1), score:  99.99%
Gradient min: -0.018941, max:  0.019518, stepsize:   1859.71
Loss: 36.9466

Iteration: 49
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.007313, max:  0.010175, stepsize:   1605.92
Loss: 39.6561

Iteration: 50
Predicted class-name: kit fox (#1), score:  99.99%
Gradient min: -0.005102, max:  0.005128, stepsize:   2222.82
Loss: 34.06

Iteration: 51
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.009598, max:  0.007205, stepsize:   1756.78
Loss: 26.9699

Iteration: 52
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006587, max:  0.006691, stepsize:   1967.08
Loss: 37.3345

Iteration: 53
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006564, max:  0.006621, stepsize:   2305.14
Loss: 39.8643

Iteration: 54
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.020868, max:  0.016884, stepsize:   1375.65
Loss: 37.7343

Iteration: 55
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005574, max:  0.005976, stepsize:   1908.25
Loss: 41.346

Iteration: 56
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.009256, max:  0.010253, stepsize:   1768.63
Loss: 35.9501
```

```
Iteration: 57
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.010851, max:  0.017633, stepsize:   1499.83
Loss: 42.5105

Iteration: 58
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005229, max:  0.006164, stepsize:   2135.08
Loss: 43.219

Iteration: 59
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006745, max:  0.006746, stepsize:   1642.55
Loss: 38.7929

Iteration: 60
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005743, max:  0.004990, stepsize:   2049.10
Loss: 45.1963

Iteration: 61
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.007454, max:  0.006493, stepsize:   1576.57
Loss: 39.2328

Iteration: 62
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005872, max:  0.006283, stepsize:   2189.59
Loss: 42.8966

Iteration: 63
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006593, max:  0.007255, stepsize:   1561.50
Loss: 43.5881

Iteration: 64
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006010, max:  0.005091, stepsize:   1858.49
Loss: 49.4166

Iteration: 65
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.011517, max:  0.009566, stepsize:   1209.07
Loss: 41.3484

Iteration: 66
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.008072, max:  0.008848, stepsize:   2159.80
Loss: 45.9205

Iteration: 67
Predicted class-name: kit fox (#1), score: 100.00%
```

```
Gradient min: -0.008305, max:  0.006268, stepsize:   1548.71
Loss: 42.3279

Iteration: 68
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006149, max:  0.009358, stepsize:   1883.51
Loss: 46.414

Iteration: 69
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006578, max:  0.006697, stepsize:   1474.94
Loss: 42.8873

Iteration: 70
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005391, max:  0.006146, stepsize:   2104.57
Loss: 49.2656

Iteration: 71
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.008280, max:  0.008504, stepsize:   1563.85
Loss: 49.4512

Iteration: 72
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.009657, max:  0.011234, stepsize:   1538.54
Loss: 52.4693

Iteration: 73
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006796, max:  0.008983, stepsize:   1630.90
Loss: 47.8848

Iteration: 74
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.007820, max:  0.007669, stepsize:   1989.92
Loss: 53.1304

Iteration: 75
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.007080, max:  0.009358, stepsize:   1601.16
Loss: 52.802

Iteration: 76
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006778, max:  0.011433, stepsize:   1852.65
Loss: 51.9139

Iteration: 77
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.007497, max:  0.007986, stepsize:   1568.70
Loss: 49.05
```

```
Iteration: 78
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005112, max:  0.006590, stepsize:   2242.10
Loss: 58.7734

Iteration: 79
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.008460, max:  0.008341, stepsize:   1525.32
Loss: 60.3876

Iteration: 80
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006530, max:  0.006501, stepsize:   1898.08
Loss: 60.4282

Iteration: 81
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.009223, max:  0.006812, stepsize:   1697.45
Loss: 59.8338

Iteration: 82
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006340, max:  0.006535, stepsize:   1737.38
Loss: 60.1702

Iteration: 83
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.008974, max:  0.008040, stepsize:   1453.65
Loss: 59.8085

Iteration: 84
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.008100, max:  0.005810, stepsize:   1709.52
Loss: 63.3426

Iteration: 85
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.006354, max:  0.008465, stepsize:   1674.72
Loss: 62.9884

Iteration: 86
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.007080, max:  0.006697, stepsize:   1713.75
Loss: 66.6128

Iteration: 87
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.015349, max:  0.009788, stepsize:   1500.05
Loss: 63.2206

Iteration: 88
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.008576, max:  0.007250, stepsize:   1536.09
```

```
Loss: 68.5237

Iteration: 89
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005157, max:  0.005224, stepsize:   1869.55
Loss: 71.2678

Iteration: 90
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.007810, max:  0.007995, stepsize:   1401.19
Loss: 62.0469

Iteration: 91
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.010371, max:  0.009543, stepsize:   1559.59
Loss: 70.518

Iteration: 92
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.009110, max:  0.006689, stepsize:   1855.04
Loss: 67.8497

Iteration: 93
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005365, max:  0.006440, stepsize:   1969.30
Loss: 69.9785

Iteration: 94
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.010603, max:  0.011318, stepsize:   1475.43
Loss: 69.6375

Iteration: 95
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.004267, max:  0.005465, stepsize:   2023.68
Loss: 76.1746

Iteration: 96
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.011737, max:  0.010223, stepsize:   1207.37
Loss: 58.1862

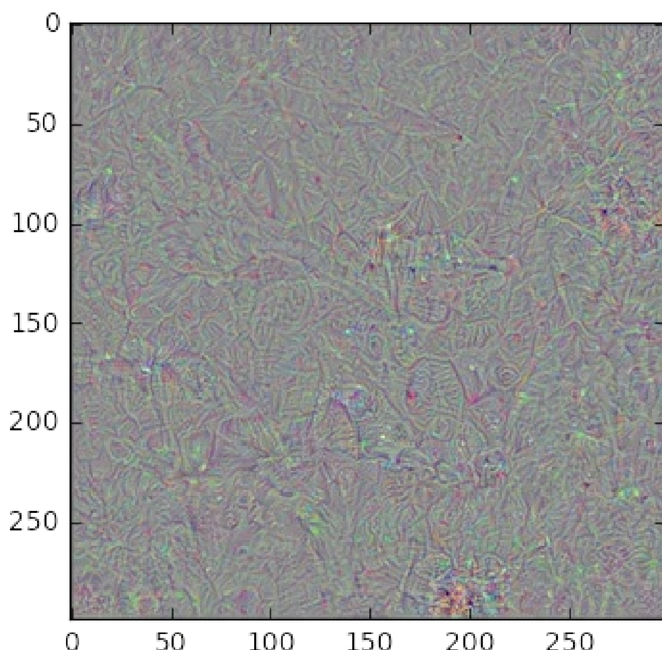
Iteration: 97
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005620, max:  0.005410, stepsize:   1992.51
Loss: 73.5772

Iteration: 98
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.007732, max:  0.010692, stepsize:   1286.44
Loss: 67.5603

Iteration: 99
```

```
Predicted class-name: kit fox (#1), score: 100.00%
Gradient min: -0.005850, max:  0.006159, stepsize:   1863.65
Loss: 75.6356
```

```
plot_image(image=image)
```



关闭TensorFlow会话

在上面使用Inception模型的函数中已经关闭了TensorFlow会话。这么做是为了节省内存，因此当计算图中添加了很多梯度函数时，电脑不会奔溃。

总结

这篇教程说明了如何优化输入图像，使得神经网络内的特征最大化。由于神经网络内给定特征（或神经元）对特定的图像反应最强烈，这让我们可以对其“喜欢看到的东西”进行可视化分析。

对神经网络的较低层，图像包含了简单的图案，比如不同类型的波浪线。随着网络越来越深，图像模式越来越复杂。我们可能会希望深层网络的模式是可识别的，比如猴子、狐狸、汽车等等，但实际上深层网络的图像模式更加复杂和抽象。

这是为什么？回想在教程 #11中，Inception模型很容易就被一些对抗噪声糊弄，而将任何输入图分类为另外的目标类别。因此，不难想象Inception模型可以识别这些在人眼看来并不清楚的抽象图像模式。可能存在无穷多的能够最大化神经网络内部特征的图像，并且人类只能识别出其中的一小部分。这也许是优化过程只找到抽象图像模式的原因。

其他方法

研究文献中还有许多指导优化过程的建议，从而找到人类更易识别的图像模式。

这篇文章提出了一种结合启发式来引导图像模式的优化过程。论文中展示了一些类别的样本图像，比如火烈鸟、鹈鹕、黑天鹅，人眼多多少少都能识别出来。在这里有方法的实现（精确的行数以后可能会改变）。这个方法需要启发式的组合并对参数进行微调，以生成这些图像。但论文中参数的选择并不明确。尽管尝试了一番，我还是无法重现他们的结果。也许我误解了这篇论文，或许启发式对他们网络架构（一种AlexNet的变体）的微调是好的，然而这篇教程中用的是更先进的Inception模型。

这篇文章提出了另一种生成人眼可识别的图像的方法。然而，实际上这个方法作弊了，因为它遍历训练集中的所有图像（比如ImageNet），找到能最大激活神经网络中给定特征的图像。然后对相似的图像做聚类和平均。将这个作为优化程序的初始图像。因此，当使用从真实照片构造的图像时，这个方法能得到更好的结果也不足为怪了。

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 尝试为网络中较低层的特征运行多次优化。得到的图像总是相同吗？
- 试着用更少或更多的优化迭代。这对图像质量有何影响？
- 试着改变卷积特征的损失函数。这可以用不同的方法来做。它将如何影响图样模式？为什么？
- 你认为优化器除了增大我们想要最大化的那个特征之外，会放大其他特征吗？你要怎么度量这个？你确定优化器一次只会最大化一个特征吗？
- 试着同时最大化多个特征。
- 在MNIST数据集上训练一个小一点的网络，然后试着对特征和层次做可视化。会更容易在图像中看到图案吗？
- 试着实现上述论文中的方法。
- 试着用你自己的方法来改善优化的图像。
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #14

DeepDream

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist / Github

介绍

在上一篇教程中，我们看到了如何用神经网络的梯度来生成图像。教程#11和#12展示了如何用梯度来生成对抗噪声 。教程#13展示了怎么用梯度来生成神经网络内部特征所响应的图像。

本文会使用一个与之前类似的方法。现在我们会用神经网络的梯度来放大输入图像中的图案（patterns）。这个通常称为DeepDream算法，但这个技术实际上有许多不同的变体。

本文基于之前的教程。你需要大概地熟悉神经网络（详见教程 #01和 #02）。

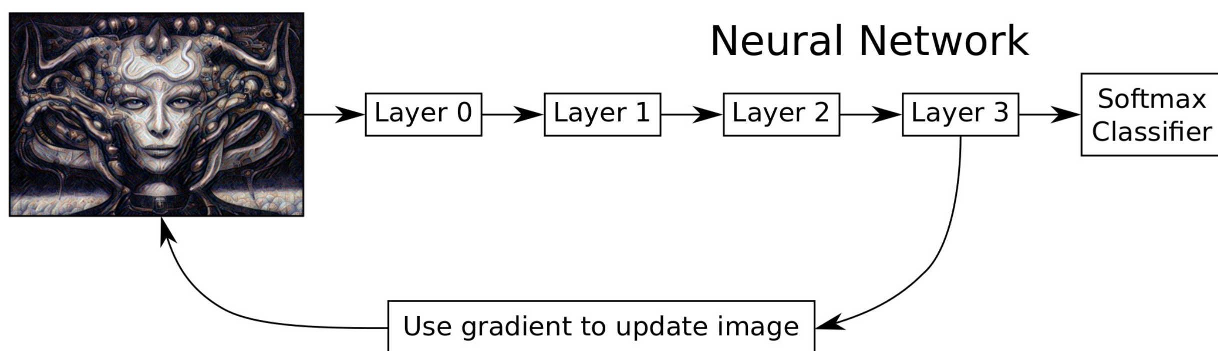
流程图

下面的流程图粗略展示了DeepDream算法的想法。我们使用的是Inception模型，它的层次要比这边显示的更多。我们使用TensorFlow自动导出网络中一个给定层相对于输入图像的梯度。然后用梯度来更新输入图像。这个过程重复多次，直到出现图案并且我们对所得到的图像满意为止。

这里的原理就是，神经网络在图像中看到一些图案的痕迹，然后我们只是用梯度把它放大了。

这里没有显示DeepDream算法的一些细节，例如梯度被平滑了，后面会讨论它的一些优点。梯度也是分块计算的，因此它可以在高分辨率的图像上工作，而不会耗尽计算机内存。

```
from IPython.display import Image, display
Image('images/14_deepdream_flowchart.png')
```



递归优化

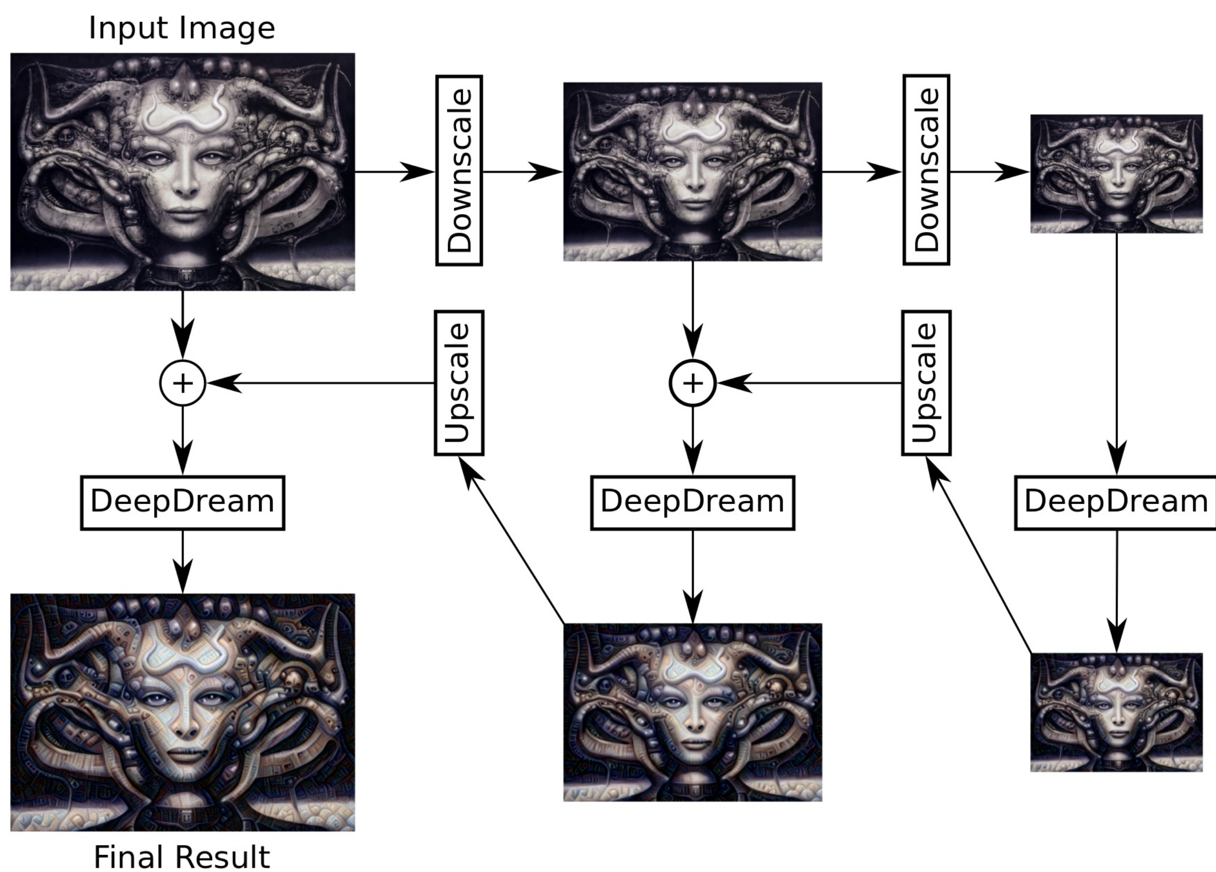
Inception模型是在相当低分辨率的图像上进行训练的，大概200-300像素。所以，当我们使用更大分辨率的图像时，DeepDream算法会在图像中创建许多小的图案。

一个解决方案是将输入图像缩小到200-300像素。但是这么低的分辨率（的结果）是像素化而且丑陋的。

另一个解决方案是多次缩小原始图像，在每个较小的图像上运行DeepDream算法。这样会在图像中创建更大的图案，然后以更高的分辨率进行改善。

这个流程图粗略显示了这个想法。算法递归地实现并且支持任何数量的缩小级别。算法有些细节并未在这里展示，比如，图像在缩小之前会做一些模糊处理，并且原始图像只是与DeepDream图像混合在一起，来增加一些原始的细节。

```
Image('images/14_deepdream_recursive_flowchart.png')
```



导入

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import random
import math

# Image manipulation.
import PIL.Image
from scipy.ndimage.filters import gaussian_filter
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'1.1.0'
```

Inception 模型

前面的一些教程都使用了Inception v3模型。本文将会使用Inception模型的另一个变体。由于Google开发者并没有很好的为其撰写文档（跟通常一样），不太清楚模型是哪个版本。我们在这里用“Inception 5h”来指代它，因为zip包的文件名就是这样，尽管看起来这是Inception模型的一个早期的、更简单的版本。

这里使用Inception 5h模型是因为它更容易使用：它接受任何尺寸的输入图像，然后创建比Inception v3模型（见教程 #13）更漂亮的图像。

```
import inception5h
```

从网上下载Inception 5h模型。这是你保存数据文件的默认文件夹。如果文件夹不存在就自动创建。

```
# inception.data_dir = 'inception/5h/'
```

如果文件夹中不存在Inception模型，就自动下载。 它有50MB。

```
inception5h.maybe_download()
```

```
Downloading Inception 5h Model ...
Data has apparently already been downloaded and unpacked.
```

载入模型，以便使用。

```
model = inception5h.Inception5h()
```

Inception 5h模型有许多层可用来做DeepDreaming。我们列出了12个最常用的层，以供参考。

```
len(model.layer_tensors)
```

```
12
```

操作图像的帮助函数

这个函数载入一张图像，并返回一个浮点型numpy数组。

```
def load_image(filename):
    image = PIL.Image.open(filename)

    return np.float32(image)
```

将图像保存成jpeg文件。图像是保存着0-255像素的numpy数组。

```
def save_image(image, filename):
    # Ensure the pixel-values are between 0 and 255.
    image = np.clip(image, 0.0, 255.0)

    # Convert to bytes.
    image = image.astype(np.uint8)

    # Write the image-file in jpeg-format.
    with open(filename, 'wb') as file:
        PIL.Image.fromarray(image).save(file, 'jpeg')
```

这是绘制图像的函数。使用matplotlib将得到低分辨率的图像。使用PIL效果比较好。

```
def plot_image(image):
    # Assume the pixel-values are scaled between 0 and 255.

    if False:
        # Convert the pixel-values to the range between 0.0 and
1.0
        image = np.clip(image/255.0, 0.0, 1.0)

        # Plot using matplotlib.
        plt.imshow(image, interpolation='lanczos')
        plt.show()
    else:
        # Ensure the pixel-values are between 0 and 255.
        image = np.clip(image, 0.0, 255.0)

        # Convert pixels to bytes.
        image = image.astype(np.uint8)

        # Convert to a PIL-image and display it.
        display(PIL.Image.fromarray(image))
```

归一化图像，则像素值在0.0到1.0之间。这个在绘制梯度时很有用。

```
def normalize_image(x):
    # Get the min and max values for all pixels in the input.
    x_min = x.min()
    x_max = x.max()

    # Normalize so all values are between 0.0 and 1.0
    x_norm = (x - x_min) / (x_max - x_min)

    return x_norm
```

对梯度做归一化之后，用这个函数绘制。

```
def plot_gradient(gradient):
    # Normalize the gradient so it is between 0.0 and 1.0
    gradient_normalized = normalize_image(gradient)

    # Plot the normalized gradient.
    plt.imshow(gradient_normalized, interpolation='bilinear')
    plt.show()
```

这个函数调整图像的大小。函数的参数是你指定的具体的图像分辨率，比如(100，200)，它也可以接受一个缩放因子，比如，参数是0.5时,图像每个维度缩小一半。

这个函数用PIL来实现，代码有点长，因为我们用numpy数组来处理图像，其中像素值是浮点值。PIL不支持这个，因此需要将图像转换成8位字节，来确保像素值在合适的范围内。然后，图像被调整大小并转换回浮点值。

```python
def resize_image(image, size=None, factor=None):
    # If a rescaling-factor is provided then use it.
    if factor is not None:
        # Scale the numpy array's shape for height and width.
        size = np.array(image.shape[0:2]) * factor

        # The size is floating-point because it was scaled.
        # PIL requires the size to be integers.
        size = size.astype(int)
    else:
        # Ensure the size has length 2.
        size = size[0:2]

    # The height and width is reversed in numpy vs. PIL.
    size = tuple(reversed(size))

    # Ensure the pixel-values are between 0 and 255.
    img = np.clip(image, 0.0, 255.0)

    # Convert the pixels to 8-bit bytes.
    img = img.astype(np.uint8)

    # Create PIL-object from numpy array.
    img = PIL.Image.fromarray(img)

    # Resize the image.
    img_resized = img.resize(size, PIL.Image.LANCZOS)

    # Convert 8-bit pixel values back to floating-point.
    img_resized = np.float32(img_resized)

    return img_resized
```

DeepDream 算法

梯度

下面的帮助函数计算了在DeepDream中使用的输入图像的梯度。Inception 5h模型可以接受任意尺寸的图像，但太大的图像可能会占用千兆字节的内存。为了使内存占用最低，我们将输入图像分割成小的图块，然后计算每小块的梯度。

然而，这可能会在DeepDream算法最终生成的图像中产生肉眼可见的线条。因此我们随机地挑选小块，这样它们的位置就是不同的。这使得在最终的DeepDream图像里，小块之间的缝隙不可见。

这个帮助函数用来确定合适的图块尺寸。比如，期望的图块尺寸为400x400像素，但实际大小取决于图像尺寸。

```
def get_tile_size(num_pixels, tile_size=400):
    """
    num_pixels is the number of pixels in a dimension of the ima
ge.
    tile_size is the desired tile-size.
    """

    # How many times can we repeat a tile of the desired size.
    num_tiles = int(round(num_pixels / tile_size))

    # Ensure that there is at least 1 tile.
    num_tiles = max(1, num_tiles)

    # The actual tile-size.
    actual_tile_size = math.ceil(num_pixels / num_tiles)

    return actual_tile_size
```

这个帮助函数计算了输入图像的梯度。图像被分割成小块，然后分别计算各个图块的梯度。图块是随机选择的，避免在最终的DeepDream图像内产生可见的缝隙。

```
def tiled_gradient(gradient, image, tile_size=400):
    # Allocate an array for the gradient of the entire image.
    grad = np.zeros_like(image)

    # Number of pixels for the x- and y-axes.
    x_max, y_max, _ = image.shape

    # Tile-size for the x-axis.
    x_tile_size = get_tile_size(num_pixels=x_max, tile_size=tile
_size)
    # 1/4 of the tile-size.
    x_tile_size4 = x_tile_size // 4

    # Tile-size for the y-axis.
    y_tile_size = get_tile_size(num_pixels=y_max, tile_size=tile
_size)
    # 1/4 of the tile-size
    y_tile_size4 = y_tile_size // 4

    # Random start-position for the tiles on the x-axis.
    # The random value is between -3/4 and -1/4 of the tile-size.

    # This is so the border-tiles are at least 1/4 of the tile-s
ize,
    # otherwise the tiles may be too small which creates noisy g
radients.
    x_start = random.randint(-3*x_tile_size4, -x_tile_size4)

    while x_start < x_max:
```

```
        # End-position for the current tile.
        x_end = x_start + x_tile_size

        # Ensure the tile's start- and end-positions are valid.
        x_start_lim = max(x_start, 0)
        x_end_lim = min(x_end, x_max)

        # Random start-position for the tiles on the y-axis.
        # The random value is between -3/4 and -1/4 of the tile-
size.
        y_start = random.randint(-3*y_tile_size4, -y_tile_size4)

        while y_start < y_max:
            # End-position for the current tile.
            y_end = y_start + y_tile_size

            # Ensure the tile's start- and end-positions are val
id.
            y_start_lim = max(y_start, 0)
            y_end_lim = min(y_end, y_max)

            # Get the image-tile.
            img_tile = image[x_start_lim:x_end_lim,
                             y_start_lim:y_end_lim, :]

            # Create a feed-dict with the image-tile.
            feed_dict = model.create_feed_dict(image=img_tile)

            # Use TensorFlow to calculate the gradient-value.
            g = session.run(gradient, feed_dict=feed_dict)

            # Normalize the gradient for the tile. This is
            # necessary because the tiles may have very different

            # values. Normalizing gives a more coherent gradient.

            g /= (np.std(g) + 1e-8)

            # Store the tile's gradient at the appropriate locat
ion.
            grad[x_start_lim:x_end_lim,
                 y_start_lim:y_end_lim, :] = g

            # Advance the start-position for the y-axis.
            y_start = y_end

        # Advance the start-position for the x-axis.
        x_start = x_end

    return grad
```

优化图像

这个函数是DeepDream算法的主要优化循环。它根据输入图像计算Inception模型中给定层的梯度。然后将梯度添加到输入图像，从而增加层张量(layer-tensor)的平均值。多次重复这个过程，并放大Inception模型在输入图像中看到的任何图案。

```
def optimize_image(layer_tensor, image,
                   num_iterations=10, step_size=3.0, tile_size=4
00,
                   show_gradient=False):
    """
    Use gradient ascent to optimize an image so it maximizes the
    mean value of the given layer_tensor.

    Parameters:
    layer_tensor: Reference to a tensor that will be maximized.
    image: Input image used as the starting point.
    num_iterations: Number of optimization iterations to perform
.
    step_size: Scale for each step of the gradient ascent.
    tile_size: Size of the tiles when calculating the gradient.
    show_gradient: Plot the gradient in each iteration.
    """

    # Copy the image so we don't overwrite the original image.
    img = image.copy()

    print("Image before:")
    plot_image(img)

    print("Processing image: ", end="")

    # Use TensorFlow to get the mathematical function for the
    # gradient of the given layer-tensor with regard to the
    # input image. This may cause TensorFlow to add the same
    # math-expressions to the graph each time this function is c
alled.
    # It may use a lot of RAM and could be moved outside the fun
ction.
    gradient = model.get_gradient(layer_tensor)

    for i in range(num_iterations):
        # Calculate the value of the gradient.
        # This tells us how to change the image so as to
        # maximize the mean of the given layer-tensor.
        grad = tiled_gradient(gradient=gradient, image=img)

        # Blur the gradient with different amounts and add
        # them together. The blur amount is also increased
        # during the optimization. This was found to give
        # nice, smooth images. You can try and change the formul
as.
```

```
            # The blur-amount is called sigma (0=no blur, 1=low blur
, etc.)
            # We could call gaussian_filter(grad, sigma=(sigma, sigm
a, 0.0))
            # which would not blur the colour-channel. This tends to
            # give psychadelic / pastel colours in the resulting ima
ges.
            # When the colour-channel is also blurred the colours of
 the
            # input image are mostly retained in the output image.
            sigma = (i * 4.0) / num_iterations + 0.5
            grad_smooth1 = gaussian_filter(grad, sigma=sigma)
            grad_smooth2 = gaussian_filter(grad, sigma=sigma*2)
            grad_smooth3 = gaussian_filter(grad, sigma=sigma*0.5)
            grad = (grad_smooth1 + grad_smooth2 + grad_smooth3)

            # Scale the step-size according to the gradient-values.
            # This may not be necessary because the tiled-gradient
            # is already normalized.
            step_size_scaled = step_size / (np.std(grad) + 1e-8)

            # Update the image by following the gradient.
            img += grad * step_size_scaled

            if show_gradient:
                # Print statistics for the gradient.
                msg = "Gradient min: {0:>9.6f}, max: {1:>9.6f}, step
size: {2:>9.2f}"
                print(msg.format(grad.min(), grad.max(), step_size_s
caled))

                # Plot the gradient.
                plot_gradient(grad)
            else:
                # Otherwise show a little progress-indicator.
                print(". ", end="")

        print()
        print("Image after:")
        plot_image(img)

        return img
```

图像递归优化

Inception模型在相当小的图像上进行训练。不清楚图像的确切大小，但可能每个维度200-300像素。如果我们使用较大的图像，比如1920x1080像素，那么上面的 `optimize_image()` 函数会在图像上添加很多小的图案。

这个帮助函数将输入图像多次缩放，然后用每个缩放图像来执行上面的 `optimize_image()` 函数。这在最终的图像中生成较大的图案。它也能加快计算速度。

```
def recursive_optimize(layer_tensor, image,
                       num_repeats=4, rescale_factor=0.7, blend=
0.2,
                       num_iterations=10, step_size=3.0,
                       tile_size=400):
    """
    Recursively blur and downscale the input image.
    Each downscaled image is run through the optimize_image()
    function to amplify the patterns that the Inception model se
es.

    Parameters:
    image: Input image used as the starting point.
    rescale_factor: Downscaling factor for the image.
    num_repeats: Number of times to downscale the image.
    blend: Factor for blending the original and processed images
.

    Parameters passed to optimize_image():
    layer_tensor: Reference to a tensor that will be maximized.
    num_iterations: Number of optimization iterations to perform
.
    step_size: Scale for each step of the gradient ascent.
    tile_size: Size of the tiles when calculating the gradient.
    """

    # Do a recursive step?
    if num_repeats>0:
        # Blur the input image to prevent artifacts when downsca
ling.
        # The blur amount is controlled by sigma. Note that the
        # colour-channel is not blurred as it would make the ima
ge gray.
        sigma = 0.5
        img_blur = gaussian_filter(image, sigma=(sigma, sigma, 0
.0))

        # Downscale the image.
        img_downscaled = resize_image(image=img_blur,
                                      factor=rescale_factor)

        # Recursive call to this function.
        # Subtract one from num_repeats and use the downscaled i
mage.
        img_result = recursive_optimize(layer_tensor=layer_tenso
r,
                                        image=img_downscaled,
                                        num_repeats=num_repeats-1
```

```
,
                                        rescale_factor=rescale_f
actor,
                                        blend=blend,
                                        num_iterations=num_itera
tions,
                                        step_size=step_size,
                                        tile_size=tile_size)

        # Upscale the resulting image back to its original size.
        img_upscaled = resize_image(image=img_result, size=image
.shape)

        # Blend the original and processed images.
        image = blend * image + (1.0 - blend) * img_upscaled

    print("Recursive level:", num_repeats)

    # Process the image using the DeepDream algorithm.
    img_result = optimize_image(layer_tensor=layer_tensor,
                                image=image,
                                num_iterations=num_iterations,
                                step_size=step_size,
                                tile_size=tile_size)

    return img_result
```

TensorFlow 会话

我们需要一个TensorFlow会话来运行图。这是一个交互式的会话，因此我们可以继续往计算图中添加梯度方程。

```
session = tf.InteractiveSession(graph=model.graph)
```

Hulk

在第一个例子中，我们有一张绿巨人的图像。注意看看DeepDream图像是如何保留绝大部分原始图像颜色的。这是由于梯度在其颜色通道中被平滑处理了，因此变得有点像灰阶的，主要改变图像的形状，而不改变其颜色。

```
image = load_image(filename='images/hulk.jpg')
plot_image(image)
```



首先，我们需要Inception模型中的张量的引用，它将在DeepDream优化算法中被最大化。在这个例子中，我们选择Inception模型的第3层（层索引2）。它有192个通道，我们将尝试最大化这些通道的平均值。

```
layer_tensor = model.layer_tensors[2]
layer_tensor
```

```
<tf.Tensor 'conv2d2:0' shape=(?, ?, ?, 192) dtype=float32>
```

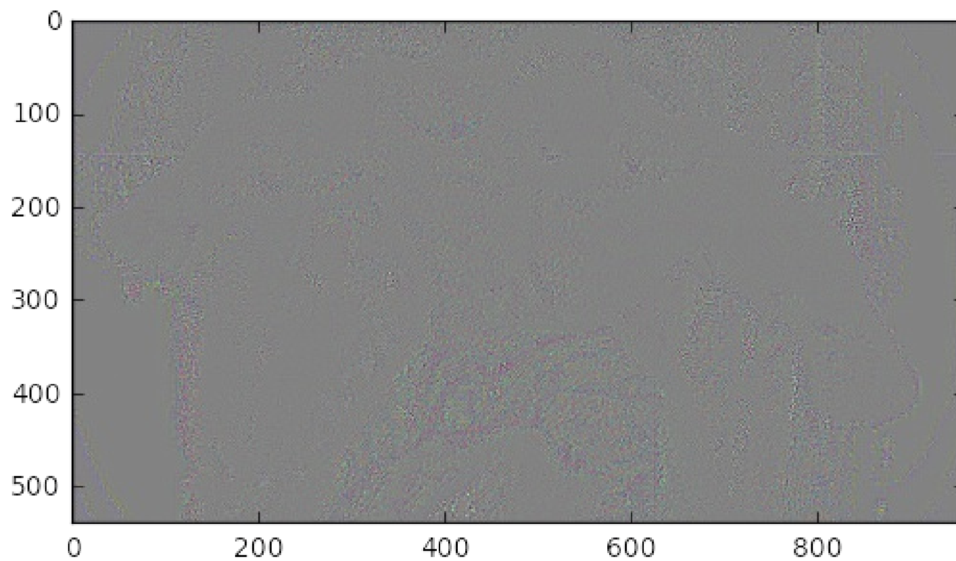
现在运行DeepDream优化算法，总共10次迭代，步长为6.0，这是下面递归优化的两倍。每次迭代我们都展示它的梯度，你可以看到图像方块之间的痕迹。

```
img_result = optimize_image(layer_tensor, image,
                   num_iterations=10, step_size=6.0, tile_size=4
00,
                   show_gradient=True)
```

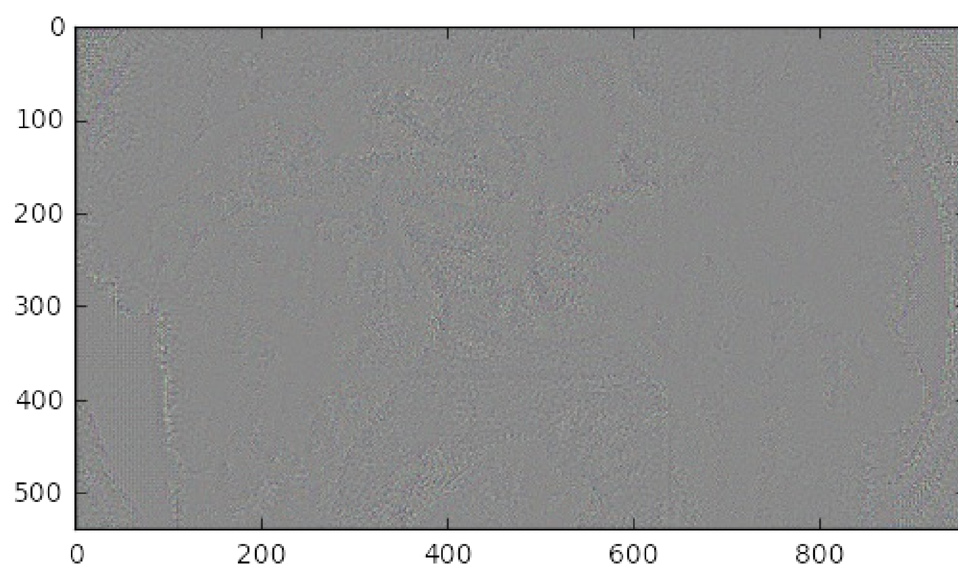
```
Image before:
```



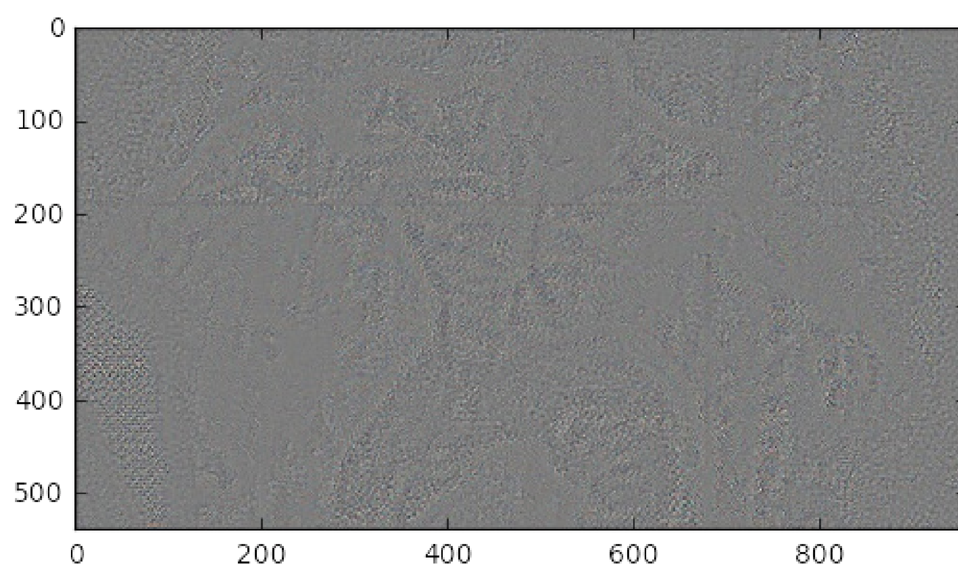
```
Processing image: Gradient min: -26.993517, max: 25.577057, step
size:      3.35
```



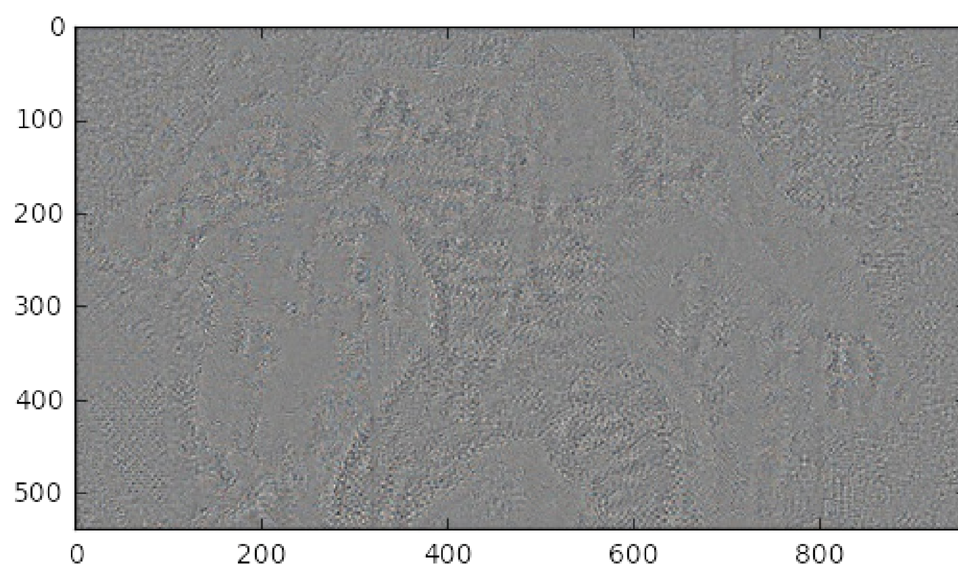
```
Gradient min: -15.383774, max: 12.962121, stepsize:      5.97
```



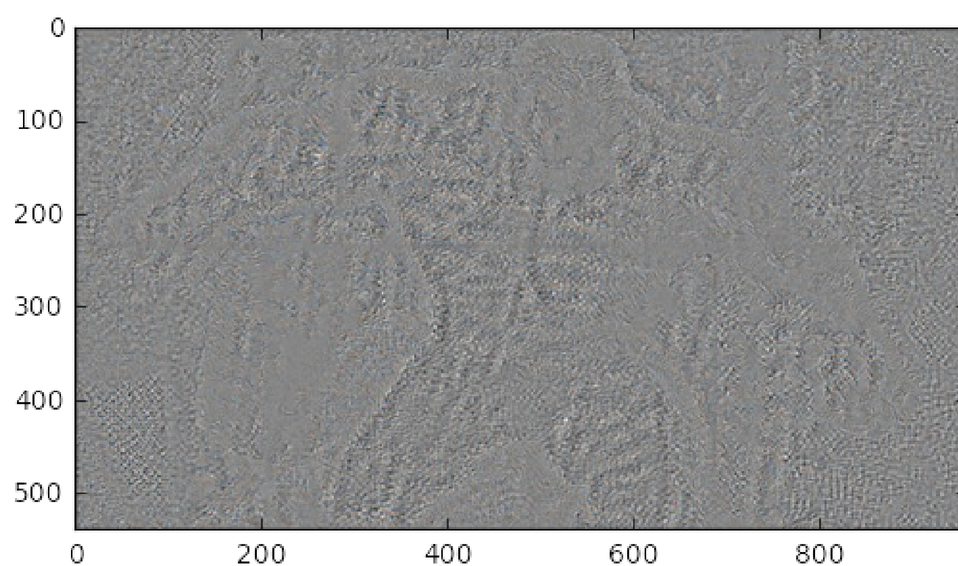

```
Gradient min: -5.993865, max:  6.191866, stepsize:     10.42
```



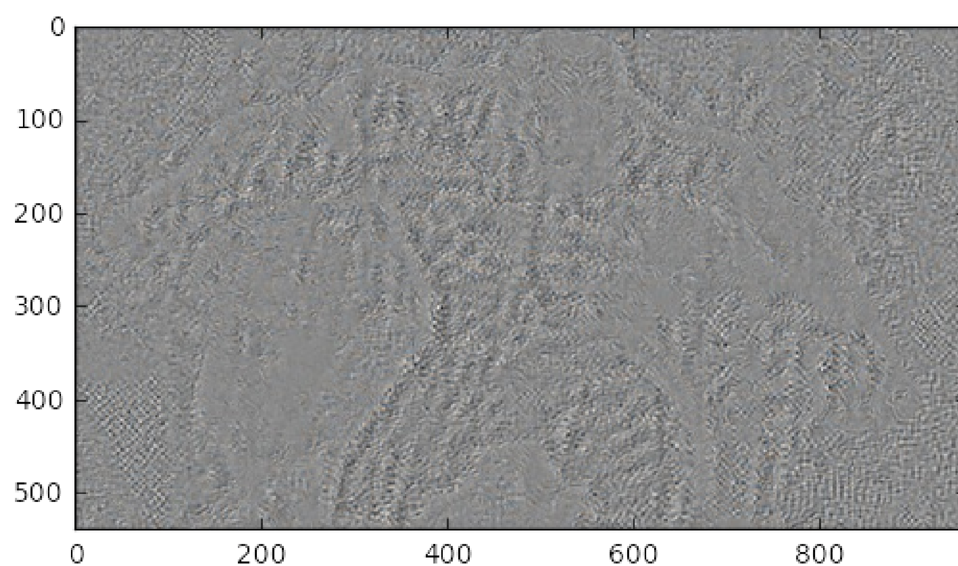

```
Gradient min: -3.638639, max:  3.307561, stepsize:     15.68
```



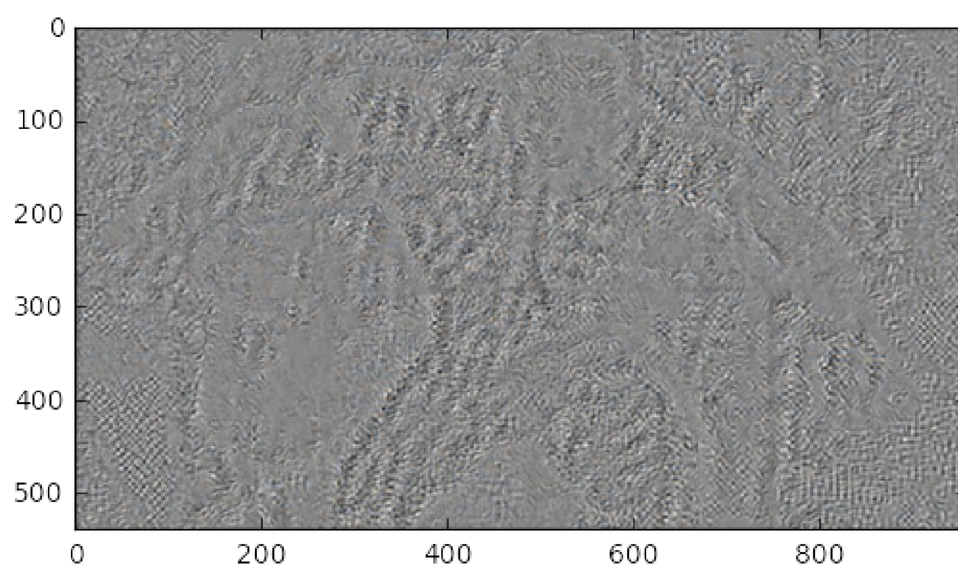

```
Gradient min: -2.407669, max:  2.166253, stepsize:     22.57
```



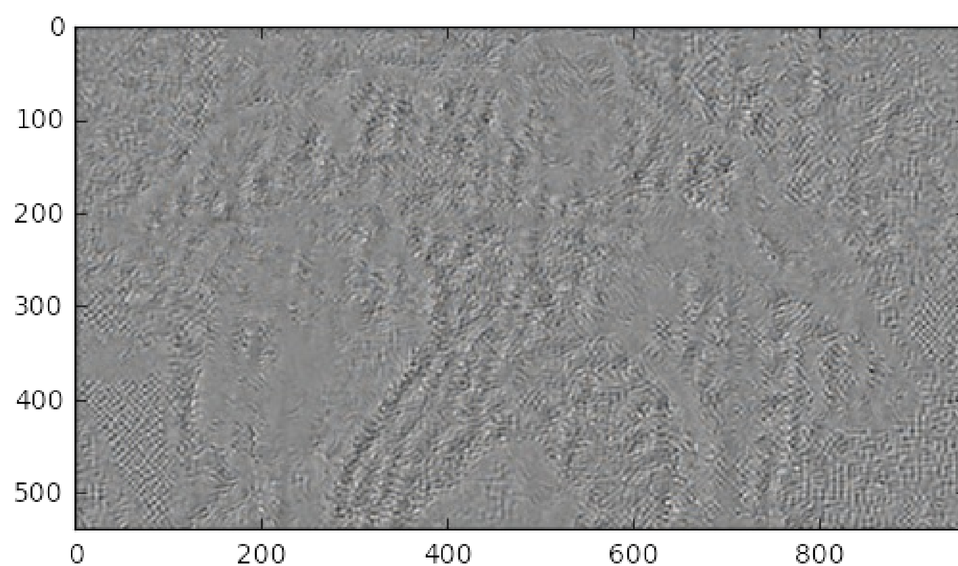

```
Gradient min: -1.716694, max:  1.467488, stepsize:     29.86
```



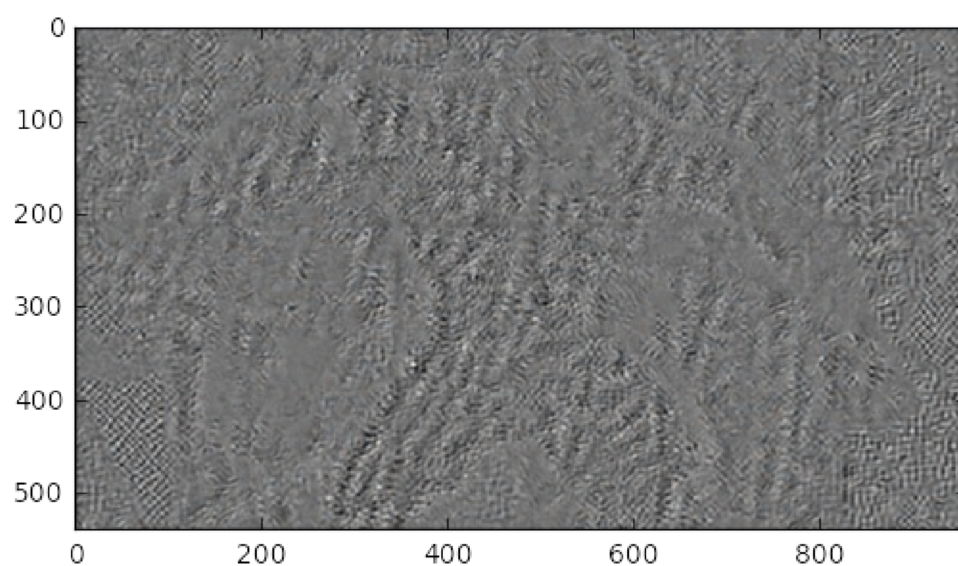
```
Gradient min: -1.153857, max:  1.025310, stepsize:     38.37
```



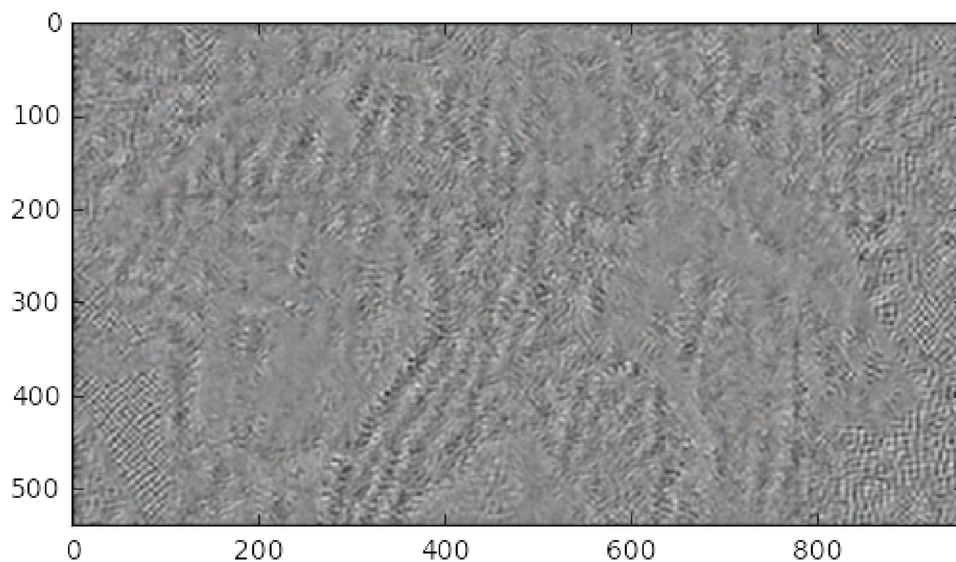
```
Gradient min: -1.026255, max:  0.869002, stepsize:     48.34
```



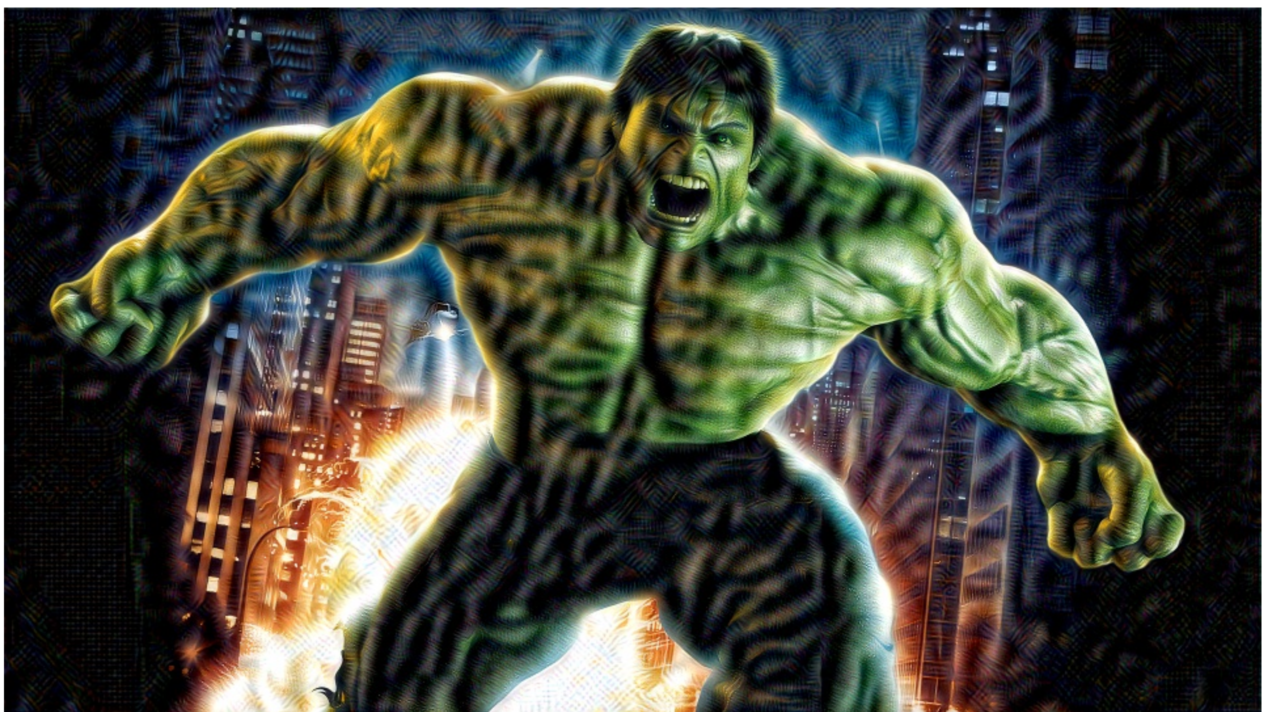

```
Gradient min: -0.634610, max:  0.765562, stepsize:     63.08
```




```
Gradient min: -0.585900, max:  0.485299, stepsize:     83.16
```




```
Image after:
```



如果你愿意的话，可以保存DeepDream图像。

```
# save_image(img_result, filename='deepdream_hulk.jpg')
```

现在，递归调用DeepDream算法。我们执行5个递归（ `num_repeats + 1` ），每个步骤中图像都被模糊并缩小，然后在缩小图像上运行DeepDream算法。接着，在每个步骤中，将产生的DeepDream图像与原始图像混合，从原始图像获取一点细节。这个过程重复了多次。

注意，现在DeepDream的图案更大了。这是因为我们先在低分辨率图像上创建图案，然后在较高分辨率图像上进行细化。

```
img_result = recursive_optimize(layer_tensor=layer_tensor, image
=image,
                 num_iterations=10, step_size=3.0, rescale_facto
r=0.7,
                 num_repeats=4, blend=0.2)
```

```
Recursive level: 0
Image before:
```



```
Processing image: . . . . . . . . . .
Image after:
```



```
Recursive level: 1
Image before:
```



```
Processing image: . . . . . . . . . . 
Image after:
```



```
Recursive level: 2
Image before:
```



```
Processing image: . . . . . . . . . . 
Image after:
```



```
Recursive level: 3
Image before:
```



```
Processing image: . . . . . . . . . . .
Image after:
```



```
Recursive level: 4
Image before:
```



```
Processing image: . . . . . . . . . .
Image after:
```



现在我们将最大化Inception模型中的较高层。使用7号层（索引6）为例。该层识别输入图像中更复杂的形状，所以DeepDream算法也将产生更复杂的图像。这一层似乎识别了狗的脸和毛发，因此DeepDream算法往图像中添加了这些东西。

再次注意，与DeepDream算法其他变体不同的是，这里输入图像的大部分颜色被保留了下来，创建了更多柔和的颜色。这是因为我们在颜色通道中平滑了梯度，使其变得有点像灰阶，因此不会太多地改变输入图像的颜色。

```
layer_tensor = model.layer_tensors[6]
img_result = recursive_optimize(layer_tensor=layer_tensor, image
=image,
                 num_iterations=10, step_size=3.0, rescale_facto
r=0.7,
                 num_repeats=4, blend=0.2)
```

下面这个例子用DeepDream算法来最大化层的特征通道的子集。此时层的索引为7，并且只有前3个特征通道被最大化。

```
layer_tensor = model.layer_tensors[7][:,:,:,0:3]
img_result = recursive_optimize(layer_tensor=layer_tensor, image
=image,
                 num_iterations=10, step_size=3.0, rescale_facto
r=0.7,
                 num_repeats=4, blend=0.2)
```

这个例子展示了最大化Inception模型最后一层的第一个特征通道的结果。不太清楚这一层及这个特征可能会在输入图像中识别出什么来。

（译者注：原文的 `num_repeates` 参数设为4，我在配有NVIDIA GT 650M的笔记本上运行程序时，会出现内存不足的情况。因此，下面将 `num_repeates` 设为3，需要的话可以自己改回来。）

```
layer_tensor = model.layer_tensors[11][:,:,:,0]
img_result = recursive_optimize(layer_tensor=layer_tensor, image
=image,
                 num_iterations=10, step_size=3.0, rescale_facto
r=0.7,
                 num_repeats=3, blend=0.2)
```

Giger

```
image = load_image(filename='images/giger.jpg')
plot_image(image)
```



```
layer_tensor = model.layer_tensors[3]
img_result = recursive_optimize(layer_tensor=layer_tensor, image
=image,
                 num_iterations=10, step_size=3.0, rescale_facto
r=0.7,
                 num_repeats=3, blend=0.2)
```

```
layer_tensor = model.layer_tensors[5]
img_result = recursive_optimize(layer_tensor=layer_tensor, image
=image,
                 num_iterations=10, step_size=3.0, rescale_facto
r=0.7,
                 num_repeats=3, blend=0.2)
```

Escher

```
image = load_image(filename='images/escher_planefilling2.jpg')
plot_image(image)
```



```
layer_tensor = model.layer_tensors[6]
img_result = recursive_optimize(layer_tensor=layer_tensor, image=image,
                 num_iterations=10, step_size=3.0, rescale_factor=0.7,
                 num_repeats=3, blend=0.2)
```

关闭TensorFlow会话

现在我们已经用TensorFlow完成了任务，关闭session，释放资源。

```
# This has been commented out in case you want to modify and experiment
# with the Notebook without having to restart it.
# session.close()
```

总结

这篇教程展示了如何使用神经网络的梯度来放大图像中的图案。输出图像似乎已经用抽象的或类似动物的图案来重新绘制了。

还有许多这种技术的变体，来生成不同的输出图像。我们鼓励你修改上述参数和算法进行实验。

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 尝试使用自己的图像。
- 试试 `optimize_image()` 和 `recursive_optimize()` 的不同参数，看看它如何影响结果。
- 试着去掉 `optimize_image()` 中的梯度。会发生什么？
- 在运行 `optimize_image()` 时绘制梯度。会看到一些失真吗？你认为是什么原因？这重要吗？你能找到一种方法来去掉它们吗？
- 尝试使用随机噪声作为输入图像。这与教程#13中用于可视化分析的类似。会生成比本教程中更好的图像吗？为什么？
- 在 `inception5h.py` 这个文件的 `Inception5h.get_gradient()` 里，删除 `tf.square()` 。DeepDream图像会发生什么变化？为什么？
- 你可以将梯度移到 `optimize_image()` 外面以节省内存吗？
- 你能使程序运行得更快吗？一个想法是直接在TensorFlow中实现高斯模糊和调整大小。
- 通过重复调用 `optimize_image()` 并在图像上放大一点，制作一个DeepDream电影。
- 逐帧处理电影。您可能需要在帧间保持稳定。
- 向朋友解释程序如何工作。

License (MIT)

TensorFlow 教程 #15

风格迁移

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube
中文翻译 thrillerist / Github

介绍

在之前的教程#14中，我们看到了如何最大化神经网络内部的特征激活，以便放大输入图像中的模式。这个称为DeepDream。

本文采用了类似的想法，不过有两张输入图：一张内容图像和一张风格图像。然后，我们希望创建一张混合图像，它包含了内容图的轮廓以及风格图的纹理。

本文基于之前的教程。你需要大概地熟悉神经网络（详见教程 #01和 #02），熟悉教程 #14中的DeepDream也很有帮助。

流程图

这张流程图显示了风格迁移算法的大体想法，尽管比起图中所展示出来的，我们所使用的VGG-16模型有更多的层次。

输入两张图像到神经网络中：一张内容图像和一张风格图像。我们希望创建一张混合图像，它包含了内容图的轮廓以及风格图的纹理。 我们通过创建几个可以被优化的损失函数来完成这一点。

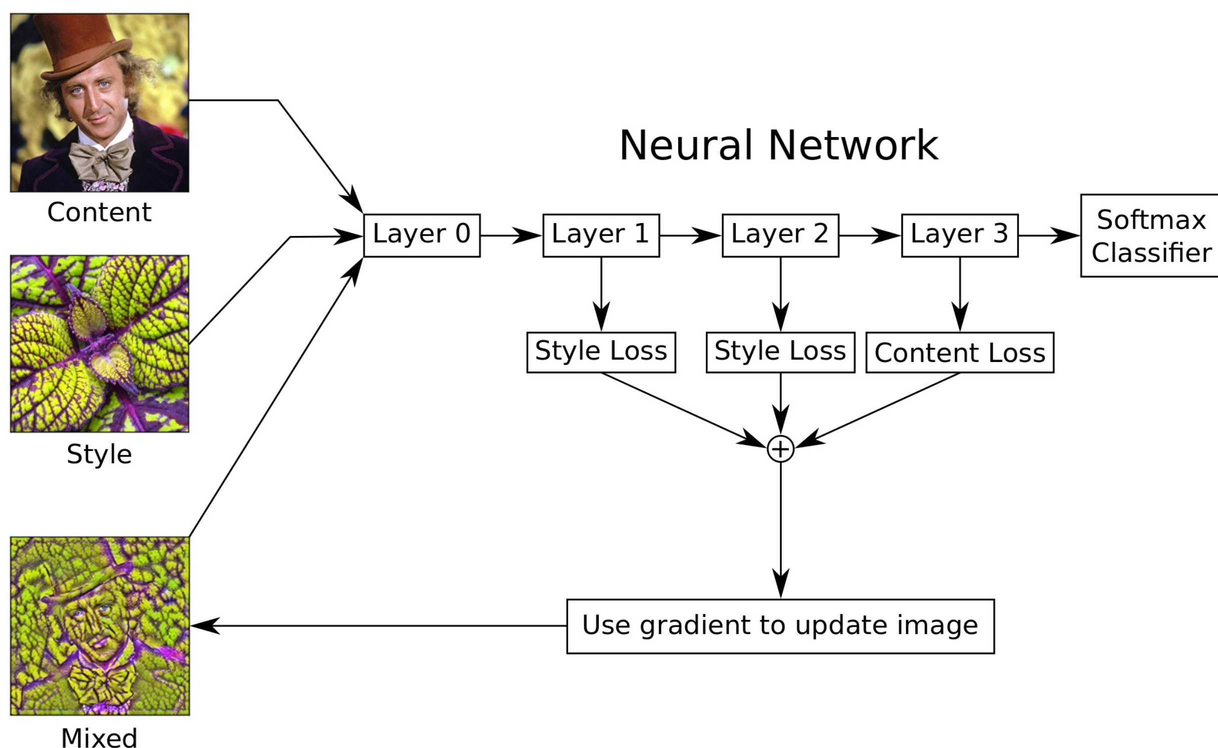
内容图像的损失函数会试着在网络的某一层或多层上，最小化内容图像以及混合图像激活特征的差距。这使得混合图像和内容图像的的轮廓相似。

风格图像的损失函数稍微复杂一些，因为它试图让风格图像和混合图像的格拉姆矩阵（Gram-matrices）的差异最小化。这在网络的一个或多个层中完成。 Gram-matrices度量了哪个特征在给定层中同时被激活。改变混合图像，使其模仿风格图像的激活模式(activation patterns)，这将导致颜色和纹理的迁移。

我们用TensorFlow来自动导出这些损失函数的梯度。然后用梯度来更新混合图像。重复多次这个过程，直到我们对结果图像满意为止。

风格迁移算法的一些细节没有在这张流程图中显示出来，比如，对于Gram-matrices的计算，计算并保存中间值来提升效率，还有一个用来给混合图像去噪的损失函数，对损失函数做归一化（normalization），这样它们更容易相对彼此缩放。

```
from IPython.display import Image, display
Image('images/15_style_transfer_flowchart.png')
```



Imports

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import PIL.Image
```

使用Python3.5.2（Anaconda）开发，TensorFlow版本是：

```
tf.__version__
```

```
'0.11.0rc0'
```

VGG-16 模型

我花了两天时间，想用之前教程#14中在DeepDream上使用的Inception 5h模型来实现风格迁移算法，但无法得到看起来足够好的图像。这有点奇怪，因为教程#14中生成的图像看起来挺好的。但回想起来，我们（在教程#14里）也用了一些技巧来得到这种质量，比如平滑梯度以及递归的降采样并处理图像。

原始论文 使用了VGG-19卷积神经网络。出于由于某些原因，对于TendorFlow来说，预训练的VGG-19模型在本教程中不够稳定。因此我们使用VGG-16模型，这是其他人制作的，可以很容易地获取并在TensorFlow中载入。方便起见，我们封装了一个类。

```
import vgg16
```

VGG-16模型是从网上下载的。这是你保存数据文件的默认文件夹。如果文件夹不存在，它就会被创建。

```
# vgg16.data_dir = 'vgg16/'
```

Download the data for the VGG-16 model if it doesn't already exist in the directory.

WARNING: It is 550 MB!

如果文件夹中没有VGG-16模型，就自动下载。

注意：它有**500MB**！

```
vgg16.maybe_download()
```

```
Downloading VGG16 Model ...
Data has apparently already been downloaded and unpacked.
```

操作图像的帮助函数

这个函数载入一张图像，并返回一个浮点型numpy数组。图像可以被自动地改变大小，因此最大的宽高等于 `max_size` 。

```python
def load_image(filename, max_size=None):
    image = PIL.Image.open(filename)

    if max_size is not None:
        # Calculate the appropriate rescale-factor for
        # ensuring a max height and width, while keeping
        # the proportion between them.
        factor = max_size / np.max(image.size)

        # Scale the image's height and width.
        size = np.array(image.size) * factor

        # The size is now floating-point because it was scaled.
        # But PIL requires the size to be integers.
        size = size.astype(int)

        # Resize the image.
        image = image.resize(size, PIL.Image.LANCZOS)

    # Convert to numpy floating-point array.
    return np.float32(image)
```

将图像保存成一个jpeg文件。给到的图像是一个包含0到255像素值的numpy数组。

```python
def save_image(image, filename):
    # Ensure the pixel-values are between 0 and 255.
    image = np.clip(image, 0.0, 255.0)

    # Convert to bytes.
    image = image.astype(np.uint8)

    # Write the image-file in jpeg-format.
    with open(filename, 'wb') as file:
        PIL.Image.fromarray(image).save(file, 'jpeg')
```

这个函数绘制出一张大的图像。给到的图像是一个包含0到255像素值的numpy数组。

```python
def plot_image_big(image):
    # Ensure the pixel-values are between 0 and 255.
    image = np.clip(image, 0.0, 255.0)

    # Convert pixels to bytes.
    image = image.astype(np.uint8)

    # Convert to a PIL-image and display it.
    display(PIL.Image.fromarray(image))
```

这个函数画出内容图像，混合图像以及风格图像。

```
def plot_images(content_image, style_image, mixed_image):
    # Create figure with sub-plots.
    fig, axes = plt.subplots(1, 3, figsize=(10, 10))

    # Adjust vertical spacing.
    fig.subplots_adjust(hspace=0.1, wspace=0.1)

    # Use interpolation to smooth pixels?
    smooth = True

    # Interpolation type.
    if smooth:
        interpolation = 'sinc'
    else:
        interpolation = 'nearest'

    # Plot the content-image.
    # Note that the pixel-values are normalized to
    # the [0.0, 1.0] range by dividing with 255.
    ax = axes.flat[0]
    ax.imshow(content_image / 255.0, interpolation=interpolation
)
    ax.set_xlabel("Content")

    # Plot the mixed-image.
    ax = axes.flat[1]
    ax.imshow(mixed_image / 255.0, interpolation=interpolation)
    ax.set_xlabel("Mixed")

    # Plot the style-image
    ax = axes.flat[2]
    ax.imshow(style_image / 255.0, interpolation=interpolation)
    ax.set_xlabel("Style")

    # Remove ticks from all the plots.
    for ax in axes.flat:
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

损失函数

这些帮助函数创建了在TensorFlow优化中用的损失函数。

这个函数创建了一个TensorFlow运算，用来计算两个输入张量的最小平均误差（Mean Squared Error）。

```
def mean_squared_error(a, b):
    return tf.reduce_mean(tf.square(a - b))
```

这个函数创建了内容图像的损失函数。它是在给定层中，内容图像和混合图像激活特征的最小平均误差。当内容损失最小时，意味着在给定层中，混合图像与内容图像的激活特征很相似。根据你所选择的层次，这会将内容图像的轮廓迁移到混合图像中。

```
def create_content_loss(session, model, content_image, layer_ids)
:
    """
    Create the loss-function for the content-image.

    Parameters:
    session: An open TensorFlow session for running the model's
graph.
    model: The model, e.g. an instance of the VGG16-class.
    content_image: Numpy float array with the content-image.
    layer_ids: List of integer id's for the layers to use in the
 model.
    """

    # Create a feed-dict with the content-image.
    feed_dict = model.create_feed_dict(image=content_image)

    # Get references to the tensors for the given layers.
    layers = model.get_layer_tensors(layer_ids)

    # Calculate the output values of those layers when
    # feeding the content-image to the model.
    values = session.run(layers, feed_dict=feed_dict)

    # Set the model's graph as the default so we can add
    # computational nodes to it. It is not always clear
    # when this is necessary in TensorFlow, but if you
    # want to re-use this code then it may be necessary.
    with model.graph.as_default():
        # Initialize an empty list of loss-functions.
        layer_losses = []

        # For each layer and its corresponding values
        # for the content-image.
        for value, layer in zip(values, layers):
            # These are the values that are calculated
            # for this layer in the model when inputting
            # the content-image. Wrap it to ensure it
            # is a const - although this may be done
```

```
            # automatically by TensorFlow.
            value_const = tf.constant(value)

            # The loss-function for this layer is the
            # Mean Squared Error between the layer-values
            # when inputting the content- and mixed-images.
            # Note that the mixed-image is not calculated
            # yet, we are merely creating the operations
            # for calculating the MSE between those two.
            loss = mean_squared_error(layer, value_const)

            # Add the loss-function for this layer to the
            # list of loss-functions.
            layer_losses.append(loss)

        # The combined loss for all layers is just the average.
        # The loss-functions could be weighted differently for
        # each layer. You can try it and see what happens.
        total_loss = tf.reduce_mean(layer_losses)

    return total_loss
```

我们将对风格层做相同的处理，但现在需要度量出哪些特征在风格层和风格图像中同时被激活，接着将这些激活模式复制到混合图像中。

一种办法是为风格层的输出张量计算一个所谓的格拉姆矩阵（Gram-matrix）。Gram-matrix本质上就是风格层中激活特征向量的点乘矩阵。

如果Gram-matrix中的一个元素的值接近于0，这意味着给定层的两个特征在风格图像中没有同时激活。反之亦然，如果Gram-matrix中有很大的值，代表着两个特征同时被激活。接着，我们会试图生成复制了风格图像激活模式的混合图像。

这个帮助函数用来计算神经网络中卷积层输出张量的Gram-matrix。真正的损失函数会在后面创建。

```
def gram_matrix(tensor):
    shape = tensor.get_shape()

    # Get the number of feature channels for the input tensor,
    # which is assumed to be from a convolutional layer with 4-d
im.
    num_channels = int(shape[3])

    # Reshape the tensor so it is a 2-dim matrix. This essential
ly
    # flattens the contents of each feature-channel.
    matrix = tf.reshape(tensor, shape=[-1, num_channels])

    # Calculate the Gram-matrix as the matrix-product of
    # the 2-dim matrix with itself. This calculates the
    # dot-products of all combinations of the feature-channels.
    gram = tf.matmul(tf.transpose(matrix), matrix)

    return gram
```

下面的函数创建了风格图像的损失函数。它和上面的 `create_content_loss()` 很像，除了我们是计算Gram-matrix而非layer输出张量的最小平方误差。

```
def create_style_loss(session, model, style_image, layer_ids):
    """
    Create the loss-function for the style-image.

    Parameters:
    session: An open TensorFlow session for running the model's
graph.
    model: The model, e.g. an instance of the VGG16-class.
    style_image: Numpy float array with the style-image.
    layer_ids: List of integer id's for the layers to use in the
 model.
    """

    # Create a feed-dict with the style-image.
    feed_dict = model.create_feed_dict(image=style_image)

    # Get references to the tensors for the given layers.
    layers = model.get_layer_tensors(layer_ids)

    # Set the model's graph as the default so we can add
    # computational nodes to it. It is not always clear
    # when this is necessary in TensorFlow, but if you
    # want to re-use this code then it may be necessary.
    with model.graph.as_default():
        # Construct the TensorFlow-operations for calculating
        # the Gram-matrices for each of the layers.
        gram_layers = [gram_matrix(layer) for layer in layers]
```

```
        # Calculate the values of those Gram-matrices when
        # feeding the style-image to the model.
        values = session.run(gram_layers, feed_dict=feed_dict)

        # Initialize an empty list of loss-functions.
        layer_losses = []

        # For each Gram-matrix layer and its corresponding value
s.
        for value, gram_layer in zip(values, gram_layers):
            # These are the Gram-matrix values that are calculat
ed
            # for this layer in the model when inputting the
            # style-image. Wrap it to ensure it is a const,
            # although this may be done automatically by TensorF
low.
            value_const = tf.constant(value)

            # The loss-function for this layer is the
            # Mean Squared Error between the Gram-matrix values
            # for the content- and mixed-images.
            # Note that the mixed-image is not calculated
            # yet, we are merely creating the operations
            # for calculating the MSE between those two.
            loss = mean_squared_error(gram_layer, value_const)

            # Add the loss-function for this layer to the
            # list of loss-functions.
            layer_losses.append(loss)

        # The combined loss for all layers is just the average.
        # The loss-functions could be weighted differently for
        # each layer. You can try it and see what happens.
        total_loss = tf.reduce_mean(layer_losses)

    return total_loss
```

下面创建了用来给混合图像去噪的损失函数。这个算法称为Total Variation Denoising，本质上就是在x和y轴上将图像偏移一个像素，计算它与原始图像的差异，取绝对值保证差异是正值，然后对整个图像所有像素求和。这个步骤创建了一个可以最小化的损失函数，用来抑制图像中的噪声。

```
def create_denoise_loss(model):
    loss = tf.reduce_sum(tf.abs(model.input[:,1:,:,:] - model.in
put[:,:-1,:,:])) + \
           tf.reduce_sum(tf.abs(model.input[:,:,1:,:] - model.in
put[:,:,:-1,:]))

    return loss
```

风格迁移算法

这是风格迁移主要的优化算法。它基本上就是在上面定义的那些损失函数上做梯度下降。

算法也使用了损失函数的归一化。这似乎是一个之前未发表过的新颖想法。在每次优化迭代中，调整损失值，使它们等于一。这让用户可以独立地设置所选风格层以及内容层的损失权重。同时，在优化过程中也修改权重，来确保保留风格、内容、去噪之间所需的比重。

```
def style_transfer(content_image, style_image,
                   content_layer_ids, style_layer_ids,
                   weight_content=1.5, weight_style=10.0,
                   weight_denoise=0.3,
                   num_iterations=120, step_size=10.0):
    """
    Use gradient descent to find an image that minimizes the
    loss-functions of the content-layers and style-layers. This
    should result in a mixed-image that resembles the contours
    of the content-image, and resembles the colours and textures
    of the style-image.

    Parameters:
    content_image: Numpy 3-dim float-array with the content-imag
e.
    style_image: Numpy 3-dim float-array with the style-image.
    content_layer_ids: List of integers identifying the content-
layers.
    style_layer_ids: List of integers identifying the style-laye
rs.
    weight_content: Weight for the content-loss-function.
    weight_style: Weight for the style-loss-function.
    weight_denoise: Weight for the denoising-loss-function.
    num_iterations: Number of optimization iterations to perform
.
    step_size: Step-size for the gradient in each iteration.
    """

    # Create an instance of the VGG16-model. This is done
    # in each call of this function, because we will add
    # operations to the graph so it can grow very large
    # and run out of RAM if we keep using the same instance.
    model = vgg16.VGG16()

    # Create a TensorFlow-session.
    session = tf.InteractiveSession(graph=model.graph)

    # Print the names of the content-layers.
    print("Content layers:")
    print(model.get_layer_names(content_layer_ids))
    print()
```

```
    # Print the names of the style-layers.
    print("Style layers:")
    print(model.get_layer_names(style_layer_ids))
    print()

    # Create the loss-function for the content-layers and -image.

    loss_content = create_content_loss(session=session,
                                       model=model,
                                       content_image=content_ima
ge,
                                       layer_ids=content_layer_i
ds)

    # Create the loss-function for the style-layers and -image.
    loss_style = create_style_loss(session=session,
                                   model=model,
                                   style_image=style_image,
                                   layer_ids=style_layer_ids)


    # Create the loss-function for the denoising of the mixed-im
age.
    loss_denoise = create_denoise_loss(model)

    # Create TensorFlow variables for adjusting the values of
    # the loss-functions. This is explained below.
    adj_content = tf.Variable(1e-10, name='adj_content')
    adj_style = tf.Variable(1e-10, name='adj_style')
    adj_denoise = tf.Variable(1e-10, name='adj_denoise')

    # Initialize the adjustment values for the loss-functions.
    session.run([adj_content.initializer,
                 adj_style.initializer,
                 adj_denoise.initializer])

    # Create TensorFlow operations for updating the adjustment v
alues.
    # These are basically just the reciprocal values of the
    # loss-functions, with a small value 1e-10 added to avoid the

    # possibility of division by zero.
    update_adj_content = adj_content.assign(1.0 / (loss_content
+ 1e-10))
    update_adj_style = adj_style.assign(1.0 / (loss_style + 1e-10
))
    update_adj_denoise = adj_denoise.assign(1.0 / (loss_denoise
+ 1e-10))

    # This is the weighted loss-function that we will minimize
    # below in order to generate the mixed-image.
    # Because we multiply the loss-values with their reciprocal
```

```
    # adjustment values, we can use relative weights for the
    # loss-functions that are easier to select, as they are
    # independent of the exact choice of style- and content-laye
rs.
    loss_combined = weight_content * adj_content * loss_content
+ \
                    weight_style * adj_style * loss_style + \
                    weight_denoise * adj_denoise * loss_denoise

    # Use TensorFlow to get the mathematical function for the
    # gradient of the combined loss-function with regard to
    # the input image.
    gradient = tf.gradients(loss_combined, model.input)

    # List of tensors that we will run in each optimization iter
ation.
    run_list = [gradient, update_adj_content, update_adj_style,
\
                update_adj_denoise]

    # The mixed-image is initialized with random noise.
    # It is the same size as the content-image.
    mixed_image = np.random.rand(*content_image.shape) + 128

    for i in range(num_iterations):
        # Create a feed-dict with the mixed-image.
        feed_dict = model.create_feed_dict(image=mixed_image)

        # Use TensorFlow to calculate the value of the
        # gradient, as well as updating the adjustment values.
        grad, adj_content_val, adj_style_val, adj_denoise_val \
        = session.run(run_list, feed_dict=feed_dict)

        # Reduce the dimensionality of the gradient.
        grad = np.squeeze(grad)

        # Scale the step-size according to the gradient-values.
        step_size_scaled = step_size / (np.std(grad) + 1e-8)

        # Update the image by following the gradient.
        mixed_image -= grad * step_size_scaled

        # Ensure the image has valid pixel-values between 0 and
255.
        mixed_image = np.clip(mixed_image, 0.0, 255.0)

        # Print a little progress-indicator.
        print(". ", end="")

        # Display status once every 10 iterations, and the last.
        if (i % 10 == 0) or (i == num_iterations - 1):
            print()
            print("Iteration:", i)
```

```
            # Print adjustment weights for loss-functions.
            msg = "Weight Adj. for Content: {0:.2e}, Style: {1:.
2e}, Denoise: {2:.2e}"
            print(msg.format(adj_content_val, adj_style_val, adj
_denoise_val))

            # Plot the content-, style- and mixed-images.
            plot_images(content_image=content_image,
                        style_image=style_image,
                        mixed_image=mixed_image)

    print()
    print("Final image:")
    plot_image_big(mixed_image)

    # Close the TensorFlow session to release its resources.
    session.close()

    # Return the mixed-image.
    return mixed_image
```

例子

这个例子展示了如何将多张图像的风格迁移到一张肖像上。

首先，我们载入内容图像，它有混合图像想要的大体轮廓。

```
content_filename = 'images/willy_wonka_old.jpg'
content_image = load_image(content_filename, max_size=None)
```

然后我们载入风格图像，它拥有混合图像想要的颜色和纹理。

```
style_filename = 'images/style7.jpg'
style_image = load_image(style_filename, max_size=300)
```

接着我们定义一个整数列表，它代表神经网络中我们用来匹配内容图像的层次。这些是神经网络层次的索引。对于VGG16模型，第5层（索引4）似乎是唯一有效的内容层。

```
content_layer_ids = [4]
```

然后，我们为风格层定义另外一个整型数组。

```
# The VGG16-model has 13 convolutional layers.
# This selects all those layers as the style-layers.
# This is somewhat slow to optimize.
style_layer_ids = list(range(13))

# You can also select a sub-set of the layers, e.g. like this:
# style_layer_ids = [1, 2, 3, 4]
```

现在执行风格迁移。它自动地为风格图像、内容图像创建合适的损失函数，然后进行多次优化迭代。这将逐步地生成一张混合图像，其拥有内容图像的大体轮廓，并且它的纹理、颜色和风格图像类似。

在CPU上这个运算会很慢！

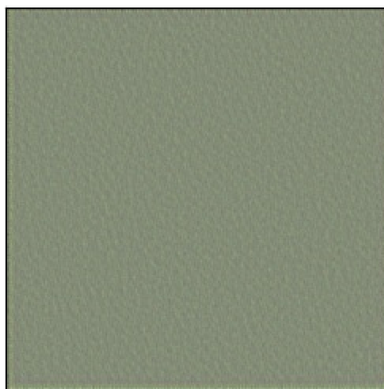
```
%%time
img = style_transfer(content_image=content_image,
                     style_image=style_image,
                     content_layer_ids=content_layer_ids,
                     style_layer_ids=style_layer_ids,
                     weight_content=1.5,
                     weight_style=10.0,
                     weight_denoise=0.3,
                     num_iterations=60,
                     step_size=10.0)
```

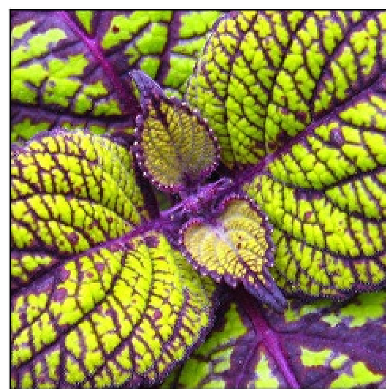
```
Content layers:
['conv3_1/conv3_1']

Style layers:
['conv1_1/conv1_1', 'conv1_2/conv1_2', 'conv2_1/conv2_1', 'conv2
_2/conv2_2', 'conv3_1/conv3_1', 'conv3_2/conv3_2', 'conv3_3/conv
3_3', 'conv4_1/conv4_1', 'conv4_2/conv4_2', 'conv4_3/conv4_3', '
conv5_1/conv5_1', 'conv5_2/conv5_2', 'conv5_3/conv5_3']

.
Iteration: 0
Weight Adj. for Content: 5.18e-11, Style: 2.14e-29, Denoise: 5.6
1e-06
```

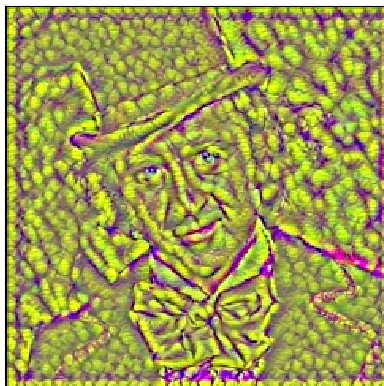

Content


Mixed


Style

```
. . . . . . . . . .
Iteration: 10
Weight Adj. for Content: 2.79e-11, Style: 4.13e-28, Denoise: 1.2
5e-07
```

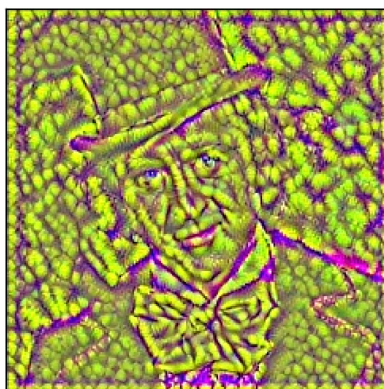

Content

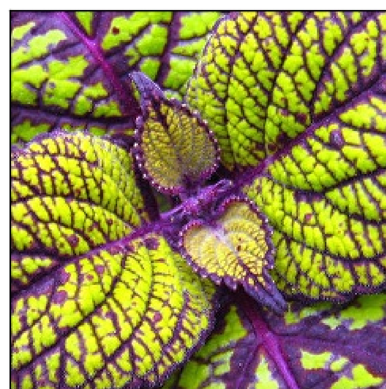

Mixed


Style

```
. . . . . . . . . .
Iteration: 20
Weight Adj. for Content: 2.63e-11, Style: 1.09e-27, Denoise: 1.3
0e-07
```


Content


Mixed


Style

```
. . . . . . . . . .
Iteration: 30
Weight Adj. for Content: 2.66e-11, Style: 1.27e-27, Denoise: 1.2
7e-07
```

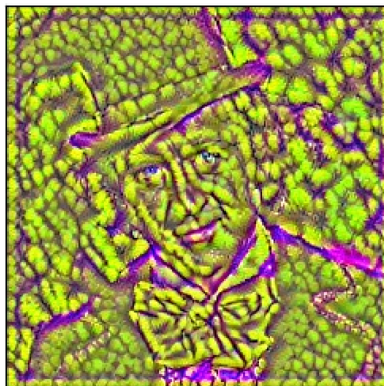

Content

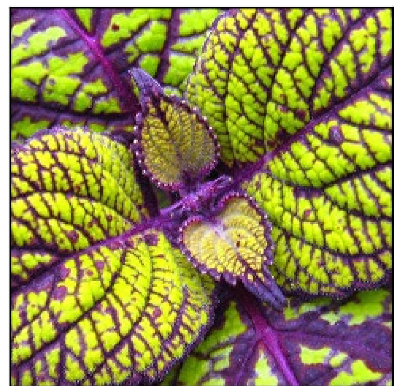

Mixed


Style

```
. . . . . . . . . .
Iteration: 40
Weight Adj. for Content: 2.73e-11, Style: 1.16e-27, Denoise: 1.2
6e-07
```


Content

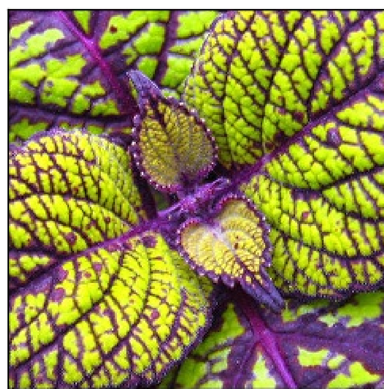

Mixed


Style

```
. . . . . . . . . .
Iteration: 50
Weight Adj. for Content: 2.75e-11, Style: 1.12e-27, Denoise: 1.2
4e-07
```


Content

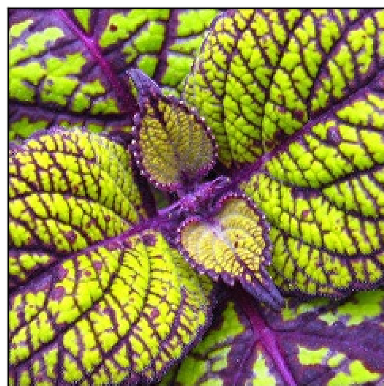

Mixed


Style

```
. . . . . . . . . .
Iteration: 59
Weight Adj. for Content: 1.85e-11, Style: 3.86e-28, Denoise: 1.0
1e-07
```


Content


Mixed


Style

```
Final image:
```



```
CPU times: user 20min 1s, sys: 45.5 s, total: 20min 46s
Wall time: 3min 4s
```

总结

这篇教程说明了用神经网络来结合两张图像内容和风格的基本想法。不幸的是，结果并不像一些商业系统那么好，比如 DeepArt，它是由这种技术的一些先驱者开发的。（结果不好的）原因暂不明确。也许我们只是需要更强的计算力，可以在高分辨率图像上以更小的步长，运行更多的优化迭代。或许我们需要更复杂的优化方法。下面的练习给出了一些可能会提升质量的建议，鼓励你尝试一下。

练习

下面使一些可能会让你提升TensorFlow技能的一些建议练习。为了学习如何更合适地使用TensorFlow，实践经验是很重要的。

在你对这个Notebook进行修改之前，可能需要先备份一下。

- 试着使用其他图像。本文中包含了一些风格图像。你可以使用自己的图像。
- 试着更多的迭代次数（比如1000-5000），以及更小的步长（比如1.0-3.0）。它会提升质量吗？* 改变风格层、内容层以及去噪时的权重。
- 试着从内容或风格图像开始优化，或许二者的平均。你可以加入一些噪声。
- 试着改变风格图和内容图的分辨率。在 `load_image()` 函数中，你可以用 `max_size` 参数来改变图像大小。它对结果有什么影响？
- 试着使用VGG-16模型的其他层。
- 改变代码，使其每10次优化迭代就保存图像。
- 在优化过程中使用常数权重。它对结果有何影响？
- 在风格层中使用不同的权重。同样，试着像其他损失函数一样自动调整权重。
- 用TensorFlow的ADAM优化器来代替基本的梯度下降。
- 使用L-BFGS优化器。目前在TensorFlow中没有实现这个。你能在风格迁移算法中使用SciPy中实现的优化器么？它有提升结果吗？
- 用另外的预训练网络，比如我们在教程 #14中使用的Inception 5h模型，或者用你从网上找到的VGG-19模型。
- 向朋友解释程序如何工作。

License (MIT)